UNIVERSIDADE DE SÃO PAULO
INSTITUTO DE MATEMÁTICA E ESTATÍSTICA
BACHARELADO EM CIÊNCIA DA COMPUTAÇÃO

MAC0499 – Trabalho de Formatura Supervisionado

Estruturas de Dados Avançadas – Algoritmos e Aplicações

Aluno: **Alessandro Bezerra da Silva**

Supervisor: **Prof. Dr. Carlos Eduardo Ferreira**

São Paulo
11 de fevereiro de 2022

RESUMO

Alessandro Bezerra da Silva. **Estruturas de Dados Avançadas – Algoritmos e Aplicações**. Monografia (Bacharelado). Instituto de Matemática e Estatística, Universidade de São Paulo, São Paulo, 2021.

Estruturas de dados estão sempre presentes, direta ou indiretamente, no cotidiano de programadores e cientistas da computação. Escolher a estrutura mais adequada para lidar com determinado tipo de problema é fundamental para a resolução do mesmo. Assim sendo, a intenção deste trabalho é estudar algumas dessas estruturas, geralmente não vistas em disciplinas obrigatórias de um curso de Ciência da Computação, a saber: Árvores B, Árvores *Splay*, Heaps Binomiais, Heaps de Fibonacci e *Tries*. Essas estruturas são muito importantes na prática. Por exemplo, a estrutura de dados Árvore B é utilizada no sistema de arquivos NTFS do Windows NT, e em sistemas gerenciadores de bancos de dados da Oracle, SQL e PostgreSQL. Também temos a estrutura de dados *Trie* sendo utilizada na implementação de dicionários.

Palavras-chave: Estruturas de dados avançadas, Árvores B, Árvores *Splay*, Heaps Binomiais, Heaps de Fibonacci, *Tries*.

ABSTRACT

Alessandro Bezerra da Silva. **Advanced Data Structures – Algorithms and Applications**. Capstone Project Report (Bachelor). Institute of Mathematics and Statistics, University of São Paulo, São Paulo, 2021.

Data structures are always present, directly or indirectly, in the daily lives of programmers and computer scientists. Choosing the most appropriate structure to deal with a particular type of problem is fundamental to solving it. Therefore, the intention of this work is to study some of these structures, generally not seen in compulsory subjects of a Computer Science course, namely: B trees, *Splay* trees, Binomial Heaps, Fibonacci Heaps and *Tries*. These structures are very important in practice. For example, a data structure B tree is used in the Windows NT NTFS file system, and in Oracle, SQL, and PostgreSQL database management systems. We also have an *Trie* data structure being used in implementing dictionaries.

Keywords: Advanced data structures, B trees, *Splay* trees, Binomial Heaps, Fibonacci Heaps, *Tries*.

SUMÁRIO

1 Introdução **1**

2 Árvores B **2**
2.1 Introdução . . . 2
2.2 Memória principal *versus* memória secundária . . . 2
2.3 Propriedades . . . 3
2.3.1 Altura de uma árvore B . . . 4
2.4 Manipulando árvores B . . . 5
2.4.1 Criando uma árvore B vazia . . . 5
2.4.2 Pesquisando uma chave . . . 5
2.4.3 Inserindo uma chave . . . 6
2.4.4 Eliminando uma chave . . . 10
2.5 Usos práticos . . . 21

3 Árvores *Splay* **22**
3.1 Introdução . . . 22
3.2 Rotações . . . 22
3.2.1 Rotações simples do tipo ZIG e ZAG . . . 22
3.2.2 Rotações duplas do tipo ZIG-ZIG e ZAG-ZAG . . . 22
3.2.3 Rotações duplas do tipo ZIG-ZAG e ZAG-ZIG . . . 23
3.3 Operações básicas . . . 23
3.3.1 Criando uma árvore *splay* . . . 23
3.3.2 Realizando uma operação *splay* . . . 24
3.3.3 Pesquisando uma chave . . . 26
3.3.4 Inserindo uma chave . . . 27
3.3.5 Removendo uma chave . . . 27
3.4 Análise amortizada . . . 27
3.5 Vantagens e desvantagens . . . 28
3.6 Usos práticos . . . 29

4 Heaps Binomiais **30**
4.1 Introdução . . . 30
4.2 Árvores binomiais . . . 30
4.2.1 Definição . . . 30
4.2.2 Propriedades . . . 31
4.3 Heaps binomiais . . . 32
4.3.1 Definição . . . 32
4.3.2 Representação . . . 32
4.4 Operações usuais . . . 33
4.4.1 Criando um heap binomial . . . 33
4.4.2 Pesquisando a chave mínima . . . 33
4.4.3 Unindo dois heaps binomiais . . . 34

4.4.4 Inserindo uma chave . . . 39
4.4.5 Extraindo o nó com a chave mínima . . . 40
4.4.6 Decrementando uma chave . . . 41
4.4.7 Eliminando uma chave . . . 42
4.5 Usos práticos . . . 42

5 Heaps de Fibonacci **43**
5.1 Introdução . . . 43
5.2 Heaps de Fibonacci . . . 43
5.2.1 Definição . . . 43
5.2.2 Função potencial . . . 44
5.2.3 Grau máximo . . . 44
5.3 Operações usuais . . . 45
5.3.1 Criando um novo heap de Fibonacci . . . 45
5.3.2 Inserindo um novo nó . . . 45
5.3.3 Encontrando o nó mínimo . . . 46
5.3.4 Extraindo o nó mínimo . . . 46
5.3.5 Unindo dois heaps de Fibonacci . . . 50
5.3.6 Decrementando uma chave . . . 50
5.3.7 Eliminando um nó . . . 53
5.4 Limitando o grau máximo de um nó . . . 53
5.5 Usos teóricos . . . 55

6 *Tries* **56**
6.1 Introdução . . . 56
6.2 Definição . . . 56
6.3 Operações usuais . . . 58
6.3.1 Criando uma *trie* . . . 58
6.3.2 Pesquisando uma chave . . . 58
6.3.3 Inserindo uma chave . . . 59
6.3.4 Removendo uma chave . . . 60
6.4 Operações especiais . . . 62
6.4.1 Obtendo todas as chaves com determinado prefixo . . . 62
6.4.2 Obtendo o maior prefixo presente numa *trie*, dada uma string . . . 63
6.4.3 Obtendo todas as chaves que verificam determinado padrão . . . 64
6.5 Usos práticos . . . 65

7 Conclusão **66**

Referências **67**

CAPÍTULO 1

INTRODUÇÃO

Lembro-me de estar em alguma aula de Estruturas de Dados II, oferecida pelo prof. Coelho em 2019, na qual se estava tentando resolver um determinado problema (que infelizmente não me lembro). A princípio, tentou-se utilizar certa estrutura de dados, porém o computador ficou processando durante vários minutos, sem retornar resposta. A seguir, observando a ineficácia da estrutura sendo exposta, o prof. Coelho substituiu tal estrutura de dados por outra, e o algoritmo retornou a resposta em questão de segundos. Aquele experimento me deixou surpreso, e entendi a importância prática de tais estruturas na tratabilidade de problemas reais. Após algum tempo, com o avanço do curso, surgiu a necessidade de se pensar em um tema pro TCC. Como esta é uma área que desperta minha curiosidade há alguns anos, decidi aprofundar os meus estudos em estruturas de dados, e documentar o meu progresso neste trabalho.

Durante os próximos capítulos, iremos estudar algumas estruturas de dados, a saber: Árvores B, Árvores *Splay*, Heaps Binomiais, Heaps de Fibonacci e *Tries*. O objetivo de cada capítulo é realizar uma breve introdução à história da estrutura, seguido pela descrição, em pseudo-código, dos algoritmos que tratam de cada operação elementar sobre a estrutura sendo estudada, como por exemplo, criação, inserções e alterações em chaves, etc. Grande parte das operações contam com o auxílio de imagens para facilitar o entendimento e, ao final do capítulo, é apresentado um pequeno conjunto de aplicações da estrutura em questão. Espera-se que o leitor adquira noções básicas sobre o funcionamento de tais estruturas.

CAPÍTULO 2
ÁRVORES B

2.1 Introdução

Desenvolvida pelos pesquisadores Rudolf Bayer e Edward McCreight no ano de 1971, enquanto trabalhavam no Boeing Scientific Research Labs, esta classe de árvores foi concebida para lidar, de forma eficiente, com grandes volumes de dados, cujo conteúdo não cabe inteiramente na memória principal do computador. Assim sendo, tal conteúdo reside na memória secundária, que é mais espaçosa e mais lenta, sendo carregado na memória principal em pedaços suficientemente pequenos, conforme forem necessários. Árvores B podem ser entendidas como uma generalização das árvores binárias balanceadas, com cada um de seus nós tendo no máximo um número predeterminado de filhos. Os nós, dentro deste contexto, são chamados de *páginas*, justamente por serem lidos e gravados em memória secundária.

Por questões históricas, a memória secundária também é chamada de *disco*. Assim, sempre que mencionarmos a palavra disco no decorrer deste texto, estaremos nos referindo à memória secundária.

2.2 Memória principal *versus* memória secundária

Um computador típico possui dois tipos de memória: principal e secundária. A memória principal é rápida e possui menor capacidade de armazenamento, enquanto a memória secundária é lenta e possui maior capacidade de armazenamento, sendo as mais comuns do tipo HDD (Hard Disk Drive) ou SSD (Solid State Drive). Memórias secundárias do tipo HDD são mais lentas que memórias secundárias do tipo SSD, pois são constituídas por partes mecânicas. As memórias secundárias do tipo SSD, por outro lado, são constituídas por *chips* de silício, o que as tornam mais rápidas que as memórias secundárias do tipo HDD. Porém, ainda assim, sua velocidade é muito inferior à velocidade da memória principal.

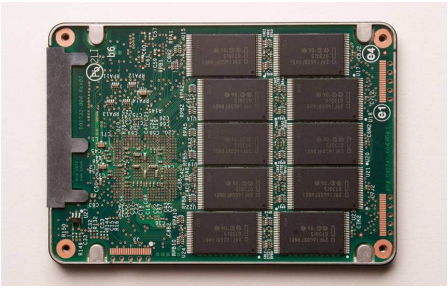

Figura 2.1: Estrutura interna do SSD.
Fonte: Olhar Digital

Figura 2.2: Estrutura interna do HDD.
Fonte: Techtudo

O tipo mais comum de memória principal em computadores atuais é o DDR (Double Data Rate), enquanto o tipo mais comum de memória secundária é o HDD, que geralmente é conectado em uma interface do tipo SATA (Serial Advanced Technology Attachment). Em sua versão mais recente, o pico da taxa de

transferência do SATA é de 6Gbits/seg, enquanto as memórias do tipo DDR superam e muito este valor, podendo chegar até centenas de gigabits por segundo, como podemos observar na tabela abaixo.

Tipo	Taxa de transferência máxima
DDR3-800	50 Gbits/seg
DDR3-1066	66.66 Gbits/seg
DDR3-1333	83.33 Gbits/seg
DDR3-1600	100 Gbits/seg

Tabela 2.1: Taxas de transferência máxima de módulos de memória do tipo DDR3. Fonte: Wikipédia (adaptado)

Como podemos observar na tabela acima, o tempo para o sistema obter certa quantidade de dados da memória secundária é muito maior que o tempo necessário para o sistema obter a mesma quantidade de dados da memória primária. Desse modo, com o objetivo de diminuir os efeitos desta diferença de tempo, e utilizando o princípio da localidade, os dados na memória secundária são divididos em páginas, com tamanho geralmente entre 2^{11} bytes e 2^{14} bytes, para que, desta forma, sempre que um dado seja requisitado do disco, uma ou mais páginas inteiras sejam trazidas à memória principal, a fim de diminuir o número de acessos à memória secundária.

Considerando-se que, na maioria dos casos, o tempo de acesso à memória secundária excede com folga o tempo necessário para processar os dados obtidos, durante o restante deste capítulo, iremos analisar os algoritmos em termos de tempo de CPU e também em termos de acessos à memória secundária. Operações de leitura e escrita de páginas em disco serão abstraídas pelas funções `DISK-READ` e `DISK-WRITE`, que recebem um ponteiro para um objeto e lêem o objeto do disco e o gravam em disco, respectivamente. Caso um objeto em particular já esteja na memória principal, uma chamada a `DISK-READ` não fará nada.

2.3 Propriedades

Uma árvore B, denotada por T, satisfaz as seguintes propriedades:

1. Possui uma raiz, denotada por `T.raiz`
2. Cada um de seus nós, denotado genericamente por x, possui:
   - Um campo `x.n`, que armazena o número de chaves presentes naquele nó.
   - `x.n` chaves, armazenadas em ordem não decrescente, em um vetor `x.chave`, tal que `x.chave[`i`]` contém a i-ésima chave, para $i \in \{1, \ldots, \texttt{x.n}\}$, e
     $$\texttt{chave[1]} \leqslant \texttt{chave[2]} \leqslant \cdots \leqslant \texttt{chave[x.n]}$$
   - `x.n` valores associados às chaves, armazenados em um vetor `x.valor`, tal que `x.valor[`i`]` está associado à chave `x.chave[`i`]`.
   - Um campo booleano `x.folha`, que assume o valor verdadeiro caso o nó seja do tipo folha, e falso caso contrário.
   - `x.n` $+ 1$ filhos, caso seja um nó interno, armazenados em um vetor `x.filho` e denotados por `x.filho[`i`]`, para $i \in \{1, \ldots, \texttt{x.n} + 1\}$. Caso x seja um nó folha, os valores para `x.filho[`i`]` são indefinidos.
3. Para cada nó interno x, seja k_i qualquer chave pertencente ao i-ésimo filho de x. As chaves de x limitam o conjunto de valores possíveis presentes nas subárvores, ou seja, valem as seguintes desigualdades:
   $$k_1 \leqslant \texttt{x.chave[1]} \leqslant k_2 \leqslant \texttt{x.chave[2]} \leqslant \cdots \leqslant \texttt{x.chave[x.n]} \leqslant k_{x.n+1}$$
4. Todos os nós folhas estão em uma mesma profundidade h.

5. Todo nó da árvore possui restrições no número de filhos. Seja t um inteiro positivo, definido de antemão, chamado ***o grau mínimo*** da árvore B. Temos que todo nó, exceto a raiz, deverá possuir no mínimo $t-1$ chaves, e todo nó, incluindo a raiz, deverá possuir no máximo $2t-1$ chaves. Ou seja, para todo nó interno x diferente da raiz

$$t-1 \leqslant \mathtt{x.n} \leqslant 2t-1$$

Além disso, caso o nó tenha exatamente $2t-1$ chaves, dizemos que o nó é ***completo***. Desse modo, todo nó interno da árvore B, com exceção da raiz, terá no mínimo t filhos e todo nó, incluindo a raiz, terá no máximo $2t$ filhos.

2.3.1 Altura de uma árvore B

O tempo de execução da maioria das operações em uma árvore B é proporcional à sua altura. Logo, é interessante encontrar um limite superior para a altura h de uma árvore B. É disso que trata o seguinte teorema.

Teorema 2.3.1. *Se $n \geqslant 1$, então a altura h de uma árvore B T de grau mínimo $t \geqslant 2$ é limitada superiormente por* $\log_t\left(\frac{n+1}{2}\right)$.

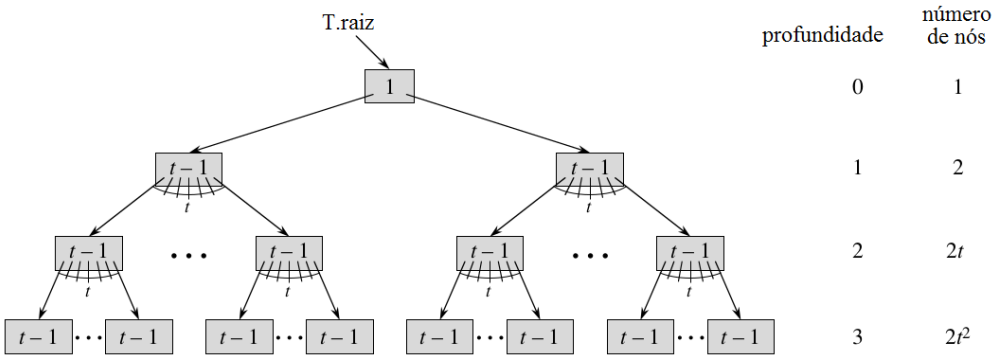

Figura 2.3: Número de nós em cada profundidade de uma árvore B, cujo número de chaves é mínimo.
Fonte: CORMEN, Thomas et al. Introduction to Algorithms, 2nd edition, The MIT Press. (adaptado)

Demonstração. Seja T' uma árvore B de altura h com o menor número de chaves possível. O número de chaves presentes em T' é minimizado quando o nó raiz possui uma única chave e todos os demais nós possuem $t-1$ chaves. Nesta situação, na profundidade 0 teremos 1 página, na profundidade 1 teremos 2 páginas, na profundidade 2 teremos $2t$ páginas, na profundidade 3 teremos $2t^2$ páginas, e assim por diante, até a profundidade h, onde teremos pelo menos $2t^{h-1}$ páginas. Denotando por n_0 o número de chaves presentes em T', temos:

$$n_0 = 1 + (t-1)\sum_{i=1}^{h} 2t^{i-1} = 1 + 2(t-1)\frac{t^h-1}{t-1} = 1 + 2(t^h-1) = 2t^h - 1$$

Uma árvore B T de altura h possuirá $n \geqslant n_0$ chaves, e desta forma vale a seguinte desigualdade, de onde segue o resultado:

$$n \geqslant n_0 = 2t^h - 1 \iff \frac{n+1}{2} \geqslant t^h \iff h \leqslant \log_t\left(\frac{n+1}{2}\right)$$

□

Desse modo, temos que $h = \mathcal{O}(\log_t n) = \mathcal{O}(\lg n)$.

2.4 Manipulando árvores B

2.4.1 Criando uma árvore B vazia

O algoritmo `B-TREE-CREATE`, ao ser invocado, retorna uma nova árvore B T vazia.

Algorithm 1

```
1: procedure B-TREE-CREATE()
2:     T ← ALLOCATE-MEMORY()
3:     x ← ALLOCATE-MEMORY()
4:     x.folha ← TRUE
5:     x.n ← 0
6:     DISK-WRITE(x)
7:     T.raiz ← x
8:     return T
```

As linhas 2-3 alocam espaço, respectivamente, para a representação de T e para a representação de x, a raiz de T. O procedimento `ALLOCATE-MEMORY` possui a finalidade de abstrair a parte que gerencia a alocação de memória. Em seguida, as linhas 4-5 inicializam adequadamente os campos de x, a linha 6 grava o novo nó recém criado em disco, a linha 7 atribui a `T.raiz` o novo nó x, e a linha 8 retorna a nova árvore B recém criada.

O consumo de tempo de `B-TREE-CREATE` é $\Theta(1)$, uma vez que as alocações de memória podem ser realizadas em tempo constante.

2.4.2 Pesquisando uma chave

O algoritmo `B-TREE-SEARCH-ON-NODE` recebe um nó x e uma chave k a ser pesquisada na subárvore com raiz em x. Retorna o par (chave, valor) em caso de sucesso na busca, e `NULL` caso contrário.

Algorithm 2

```
1: procedure B-TREE-SEARCH-ON-NODE(x, k)
2:     i ← 1
3:     while i ⩽ x.n and k > x.chave[i] do
4:         i ← i + 1
5:     if i ⩽ x.n and k = x.chave[i] then
6:         return (k, x.valor[i])
7:     if x.folha then
8:         return NULL
9:     else
10:        DISK-READ(x.filho[i])
11:        return B-TREE-SEARCH(x.filho[i], k)
```

Usando o fato das chaves em x estarem em ordem não decrescente, as linhas 2-4 procuram pela primeira posição i no vetor de chaves tal que $k \leqslant$ `x.chave[`i`]`. Após a execução destas linhas, apenas uma destas situações pode ocorrer:

- $k <$ `x.chave[`i`]`: se x for um nó interno, então a busca deve continuar na subárvore `x.filho[`i`]` (linhas 10-11), pois toda chave em `x.filho[`i`]` é menor do que `x.chave[`i`]`, pelas propriedades da árvore B. Por outro lado, se x for um nó folha, então a busca foi malsucedida, e o algoritmo devolve `NULL` (linhas 7-8).
- $k =$ `x.chave[`i`]` (linha 5), então a linha 6 retorna o respectivo par (chave, valor).
- $i >$ `x.n`, então a busca continua em `x.filho[x.n` $+ 1$`]`, já que todas as chaves em x são menores do que k, e tal chave, se estiver presente em algum lugar, pelas propriedades da árvore B, só poderá estar nesta subárvore. As linhas 9-11 tratam este caso.

Por fim, a função abaixo é um *wrapper*, ou seja, recebe uma árvore B T, uma chave k a ser pesquisada e apenas prepara o terreno para chamarmos a função `B-TREE-SEARCH-ON-NODE` na raiz de T.

Algorithm 3

```
1: procedure B-TREE-SEARCH(T, k)
2:     x ← T.raiz
3:     DISK-READ(x)
4:     return B-TREE-SEARCH-ON-NODE(x, k)
```

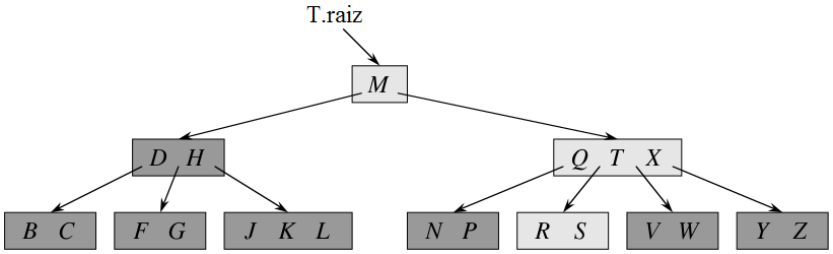

Figura 2.4: Exemplo de busca em uma árvore B. A chave R foi pesquisada, e o caminho percorrido está destacado em claro. Fonte: CORMEN, Thomas et al. Introduction to Algorithms, 2nd edition, The MIT Press. (adaptado)

Uma busca pode ter que descer por toda a árvore B, sendo que, a cada nó visitado, uma leitura de disco foi realizada previamente. Assim sendo, o número de acessos ao disco é $\mathcal{O}(h) = \mathcal{O}(\log_t n)$. Além disso, para cada nó visitado, `x.n` vale no máximo $2t - 1$; portanto o loop *while* de `B-TREE-SEARCH-ON-NODE` é executado não mais que $2t - 1$ vezes. Logo, o tempo de CPU consumido no pior caso é $\mathcal{O}((2t-1) \cdot h) = \mathcal{O}(t \cdot h) = \mathcal{O}(t \cdot \log_t n)$. Como t é definido de antemão, temos que o tempo de CPU consumido no pior caso, bem como o número de acessos ao disco, são $\mathcal{O}(\lg n)$.

2.4.3 Inserindo uma chave

A inserção de novas chaves em uma árvore B ocorre em seus nós folhas. Porém, não poderíamos simplesmente criar um novo nó folha e inserí-lo na árvore B, pois isso violaria algumas das propriedades discutidas anteriormente. Assim sendo, durante a inserção, quando nos depararmos com um nó folha x que já está cheio, iremos dividí-lo em dois outros nós, em torno de sua mediana $m =$ `x.chave[`t`]`. O primeiro deles conterá os $t - 1$ elementos menores do que m, e o segundo conterá os $t - 1$ elementos maiores do que m. A mediana m passará a pertencer ao pai de x. Caso o pai de x também fique cheio, o procedimento se repete, até que a estrutura volte a respeitar a todas as propriedades de uma árvore B.

Porém, ao invés de inserir uma chave em um nó possivelmente cheio e realizar as divisões necessárias recursivamente de modo ascendente, podemos inserir uma nova chave k em um nó folha percorrendo a árvore uma única vez, tornando nosso algoritmo mais eficiente, já que o número de acessos ao disco será reduzido. Isto se dará da seguinte maneira: conforme descemos pela árvore B, durante o caminho, sempre que descobrirmos um nó que já está cheio, realizaremos uma operação de divisão, obtendo assim dois nós, e continuaremos descendo recursivamente. Consequentemente, para cada nó x atingido pela recursão, sempre teremos a certeza de que o pai de x não é completo.

O procedimento `B-TREE-SPLIT-CHILD` abaixo concretiza a operação de divisão mencionada. Tal procedimento recebe um nó x não completo, um outro nó y completo e um índice i tal que `x.filho[`i`]` $= y$. A função transfere as $t - 1$ maiores chaves de y para um novo nó z, transfere a mediana m de y para x e atribui z como filho de x.

Algorithm 4

```
1:  procedure B-TREE-SPLIT-CHILD(x, y, i)
2:      z ← ALLOCATE-MEMORY()
3:      z.folha ← y.folha
4:      z.n ← t - 1
5:      for j ← 1 to t - 1 do
6:          z.chave[j] ← y.chave[t + j]
7:          z.valor[j] ← y.valor[t + j]
8:      if not y.folha then
9:          for j ← 1 to t do
10:             z.filho[j] ← y.filho[t + j]
11:     y.n ← t - 1
12:     for j ← x.n + 1 downto i + 1 do
13:         x.filho[j + 1] ← x.filho[j]
14:     x.filho[i + 1] ← z
15:     for j ← x.n downto i do
16:         x.chave[j + 1] ← x.chave[j]
17:         x.valor[j + 1] ← x.valor[j]
18:     x.chave[i] ← y.chave[t]
19:     x.valor[i] ← y.valor[t]
20:     x.n ← x.n + 1
21:     DISK-WRITE(x)
22:     DISK-WRITE(y)
23:     DISK-WRITE(z)
```

As linhas 2-4 do algoritmo `B-TREE-SPLIT-CHILD` criam um novo nó z, que conterá as $t-1$ maiores chaves de y. Em seguida, as linhas 5-7 copiam tais chaves para o nó z, incluindo seus respectivos valores. Se y é um nó interno, então as linhas 8-10 copiam os t últimos filhos de y para z. A linha 11 atribui a `y.n` o número de chaves restantes no nó y, as linhas 12-13 abrem espaço no vetor de filhos do nó x para a inserção do novo filho z, a linha 14 atribui z como filho de x, as linhas 15-17 abrem espaço para a inserção da chave mediana m, outrora pertencente a y, e a linhas 18-19 inserem a mediana em x, com seu respectivo valor associado. A linha 20 incrementa o número de chaves no nó x, e as linhas 21-23 gravam todas as alterações em disco.

O loop *for* da linha 5 é executado $t-1$ vezes, o loop *for* da linha 9 é executado t vezes (caso o nó y seja interno) e os loops *for* das linhas 12 e 15 são executados $\texttt{x.n} - i + 1 \leqslant 2t$ vezes. Logo, como todos os loops *for* são executados $\mathcal{O}(t)$ vezes, com t definido de antemão, temos que a complexidade de tempo para `B-TREE-SPLIT-CHILD` é $\Theta(1)$.

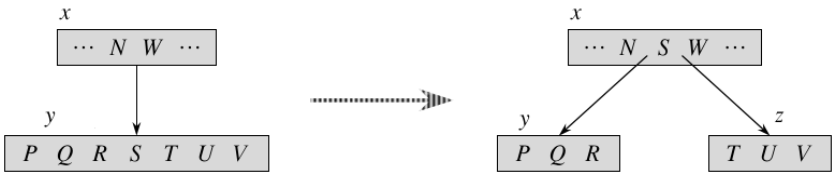

Figura 2.5: Divisão de um nó, para $t = 4$. O nó y foi dividido em dois nós y e z, e a chave mediana foi movida para x. Fonte: CORMEN, Thomas et al. Introduction to Algorithms, 2nd edition, The MIT Press. (adaptado)

De posse do procedimento `B-TREE-SPLIT-CHILD`, podemos introduzir o procedimento `B-TREE-INSERT`, que recebe uma árvore B T, uma chave k, um valor v associado à chave k e os inserem em T.

Algorithm 5

```
1:  procedure B-TREE-INSERT(T, k, v)
2:      r ← T.raiz
3:      DISK-READ(r)
4:      if r.n = 2t - 1 then
5:          s ← ALLOCATE-MEMORY()
6:          T.raiz ← s
7:          s.folha ← FALSE
8:          s.n ← 0
9:          s.filho[1] ← r
10:         B-TREE-SPLIT-CHILD(s, r, 1)
11:         B-TREE-INSERT-NONFULL(s, k, v)
12:     else
13:         B-TREE-INSERT-NONFULL(r, k, v)
```

Se a raiz r de T é completa (linha 4), então será criado um novo nó s, que passará a ser a raiz de T, tendo o nó r como único filho (linhas 5-9). Em seguida, será realizada a divisão do nó r (linha 10), e por fim será invocada a função `B-TREE-INSERT-NONFULL`(s, k, v) com segurança na linha 11 (O procedimento `B-TREE-INSERT-NONFULL` será vista com detalhes em breve; tal procedimento supõe que s é um nó não completo). Por outro lado, se a raiz r não é completa, então o método `B-TREE-INSERT-NONFULL` é invocado tomando como argumentos r, k e v.

Como a inserção de uma nova chave é realizada em um nó folha, O procedimento `B-TREE-INSERT` procura pelo nó folha apropriado para a inserção de k, usando o algoritmo `B-TREE-INSERT-NONFULL`. Cada chamada recursiva consome tempo $\Theta(1)$, e assim o consumo de tempo para a inserção de um elemento na árvore B T é $\Theta(h) = \Theta(\log_t n) = \Theta(\lg n)$.

Finalmente, o algoritmo `B-TREE-INSERT-NONFULL` recebe como argumentos um nó x, sob a hipótese de x não ser completo, uma chave k e um valor v associado à chave k, a serem inseridos na subárvore com raiz em x.

Algorithm 6

```
1:  procedure B-TREE-INSERT-NONFULL(x, k, v)
2:      i ← x.n
3:      if x.folha then
4:          while i ⩾ 1 and k < x.chave[i] do
5:              x.chave[i + 1] ← x.chave[i]
6:              x.valor[i + 1] ← x.valor[i]
7:              i ← i - 1
8:          x.chave[i + 1] ← k
9:          x.valor[i + 1] ← v
10:         x.n ← x.n + 1
11:         DISK-WRITE(x)
12:     else
13:         while i ⩾ 1 and k < x.chave[i] do
14:             i ← i - 1
15:         i ← i + 1
16:         DISK-READ(x.filho[i])
17:         if x.filho[i].n = 2t - 1 then
18:             B-TREE-SPLIT-CHILD(x, x.filho[i], i)
19:             if k > x.chave[i] then
20:                 i ← i + 1
21:         B-TREE-INSERT-NONFULL(x.filho[i], k, v)
```

Se x for um nó folha (linha 3), então as linhas 4-7 abrem espaço nos vetores de chaves e valores para a inserção da nova chave k e do novo valor v, respeitando a ordem não decrescente das chaves. As

linhas 8-9 inserem a chave k e o valor v em x, a linha 10 incrementa o número chaves presentes no nó x e a linha 11 grava as alterações em disco. Por outro lado, se x for um nó interno (linha 12), então as linhas 13-15 encontram o filho `x.filho[`i`]` adequado para a inserção de k. Se esse filho estiver cheio (linha 17), então é realizada uma operação de divisão nele (linha 18), a fim de garantir a hipótese do procedimento `B-TREE-INSERT-NONFULL` ser chamado apenas em nós não completos. As linhas 19-20 decidem, após a operação de divisão, em qual subárvore a recursão deverá seguir, e a linha 21 continua recursivamente o procedimento de inserção.

Cada chamada recursiva a `B-TREE-INSERT-NONFULL` consome tempo $\mathcal{O}(t)$, pois os loops do procedimento são executados um número de vezes limitado por $2t$. Como `B-TREE-INSERT-NONFULL` deve descer até um nó folha, que está em uma profundidade h, temos que o consumo de tempo total para `B-TREE-INSERT-NONFULL` é $\mathcal{O}(t \cdot \log_t n)$, e o número total de acessos ao disco é $\mathcal{O}(\log_t n)$, pois cada encarnação de `B-TREE-INSERT-NONFULL` realiza uma operação `DISK-READ`, que possui tempo constante. Mas t é definido de antemão, então o número de acessos ao disco, bem como o consumo de tempo total, são $\mathcal{O}(\lg n)$.

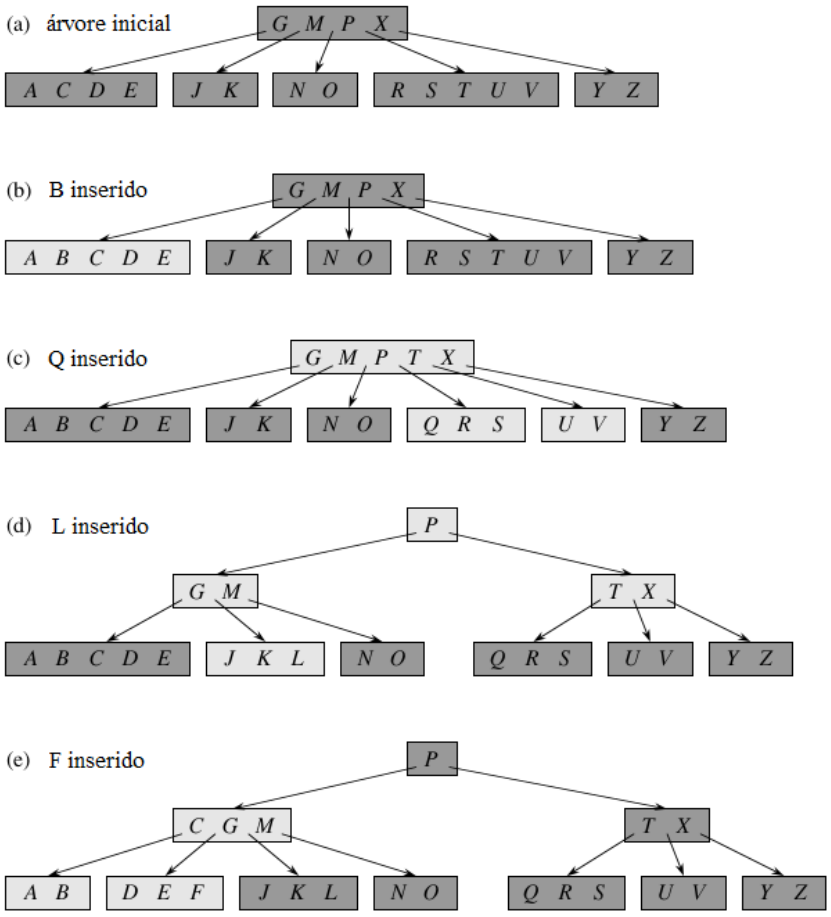

Figura 2.6: Inserção de chaves em uma árvore B, com grau mínimo $t = 3$. Em **(a)**, a árvore B em seu estado inicial. Em **(b)**, a inserção da chave B em um nó não completo ACDE. Em **(c)**, a inserção da chave Q em um nó completo RSTUV. Observe que a chave T passou a pertencer ao nó GMPX, após a operação de divisão. Em **(d)**, a inserção da chave L no nó não completo JK. Finalmente, em **(e)**, a inserção de F no nó completo ABCDE. Observe novamente a operação de divisão acontecendo. A chave C passou a pertencer ao nó GM. Fonte: CORMEN, Thomas et al. Introduction to Algorithms, 2nd edition, The MIT Press. (adaptado)

2.4.4 Eliminando uma chave

Antes de estudarmos o algoritmo que faz a remoção da chave propriamente dita, precisamos entender o funcionamento de algumas funções auxiliares, que nos ajudarão na remoção de uma chave k, e seu valor associado v, de uma árvore B T, de tal forma que, ao final do procedimento, as propriedades de uma árvore B continuem sendo satisfeitas.

O algoritmo `ROTATE-LEFT` abaixo recebe como parâmetros páginas x, y, z e um índice i tal que `x.filho[`i`]` $= y$ e `x.filho[`$i+1$`]` $= z$. A finalidade do algoritmo é aumentar em uma unidade o número de chaves em y, e diminuir em uma unidade o número de chaves em z, transferindo adequadamente as chaves entre as páginas, de tal modo que as propriedades de uma árvore B continuem sendo satisfeitas.

Algorithm 7

```
1:  procedure ROTATE-LEFT(x, y, z, i)
2:      y.chave[y.n + 1] ← x.chave[i]
3:      y.valor[y.n + 1] ← x.valor[i]
4:      x.chave[i] ← z.chave[1]
5:      x.valor[i] ← z.valor[1]
6:      y.n ← y.n + 1
7:      for j = 1 to z.n - 1 do
8:          z.chave[j] ← z.chave[j + 1]
9:          z.valor[j] ← z.valor[j + 1]
10:     if not z.folha then
11:         y.filho[y.n + 1] ← z.filho[1]
12:         for j = 1 to z.n do
13:             z.filho[j] ← z.filho[j + 1]
14:     z.n ← z.n - 1
15:     DISK-WRITE(x)
16:     DISK-WRITE(y)
17:     DISK-WRITE(z)
```

As linhas 2-3 do algoritmo `ROTATE-LEFT` inserem a chave `x.chave[`i`]` e o valor `x.valor[`i`]` na página y (note que não podemos simplesmente remover uma chave de z e seu valor associado e os inserir em y, pois isto violaria a propriedade 3 na definição de árvore B). Em seguida, as linhas 4-5 alteram o valor de `x.chave[`i`]` e `x.valor[`i`]` para `z.chave[1]` e `z.valor[1]`, respectivamente, e a linha 6 atualiza o número de chaves presentes na página y. Em seguida, o loop das linhas 7-9 corrige a posição das chaves e valores remanescentes na página z. Caso tal página não seja do tipo folha (linha 10), então a linha 11 torna `z.filho[1]` o último filho de y, e em seguida o loop das linhas 12-13 corrige a posição dos ponteiros para os filhos de z que restaram. Finalmente, a linha 14 acerta o número de elementos presentes na página z, e as linhas 15-17 gravam todas as alterações em disco.

Como os loops nas linhas 7-9 e 12-13 são executados na ordem de `z.n` vezes, e tal valor é limitado por $2t-1$ (constante, pois t é definido de antemão), temos que a complexidade de tempo do nosso algoritmo `ROTATE-LEFT` é $\Theta(1)$.

Figura 2.7: Operação de rotação à esquerda. O nó `BCDE` recebeu de seu pai `AFLOU` a chave `F`; seu pai, por sua vez, recebeu a chave `G` do nó `GHIJK`.

Nosso próximo algoritmo a ser analisado, `ROTATE-RIGHT`, possui funcionamento muito parecido com

o algoritmo anterior. Recebe os mesmos parâmetros x, y, z e i mencionados anteriormente, tal que `x.filho[i]` $= y$ e `x.filho[i+1]` $= z$. O objetivo desta função é aumentar em uma unidade o número de chaves em z, e diminuir em uma unidade o número de chaves em y, de tal modo que as propriedades de uma árvore B continuem sendo satisfeitas.

Algorithm 8

```
1:  procedure ROTATE-RIGHT(x, y, z, i)
2:      for j = z.n downto 1 do
3:          z.chave[j + 1] ← z.chave[j]
4:          z.valor[j + 1] ← z.valor[j]
5:      z.n ← z.n + 1
6:      z.chave[1] ← x.chave[i]
7:      z.valor[1] ← x.valor[i]
8:      x.chave[i] ← y.chave[y.n]
9:      x.valor[i] ← y.valor[y.n]
10:     if not z.folha then
11:         for j = z.n downto 1 do
12:             z.filho[j + 1] ← z.filho[j]
13:         z.filho[1] ← y.filho[y.n + 1]
14:     y.n ← y.n - 1
15:     DISK-WRITE(x)
16:     DISK-WRITE(y)
17:     DISK-WRITE(z)
```

O loop das linhas 2-4 abre espaço para a inserção de uma nova chave e um novo valor em z, que obrigatoriamente deverão estar no início dos vetores `z.chave` e `z.valor`, respectivamente, para que as propriedades da árvore B continuem sendo satisfeitas. A linha 5 atualiza o contador de chaves em z, e as linhas 6-7 atualizam o valor de `z.chave[1]` e `z.valor[1]` para `x.chave[i]` e `x.valor[i]`, respectivamente. Por sua vez, as linhas 8-9 atribuem a `x.chave[i]` e `x.valor[i]` o conteúdo de `y.chave[y.n]` e `y.valor[y.n]`, respectivamente, e caso o nó z não seja do tipo folha (linha 10), o loop das linhas 11-12 e a linha 13 atualizam a posição dos ponteiros adequadamente, abrindo espaço no início do vetor `z.filho` para a inserção do ponteiro `y.filho[y.n + 1]`. Em seguida, a linha 14 atualiza o número de chaves presentes em y, e as linhas 15-17 gravam todas as alterações em disco.

Os loops nas linhas 2-4 e 11-12 em `ROTATE-RIGHT` são executados na ordem de `z.n` vezes, sendo este valor limitado por $2t - 1$. Assim sendo, a complexidade de tempo para `ROTATE-RIGHT` também é $\Theta(1)$.

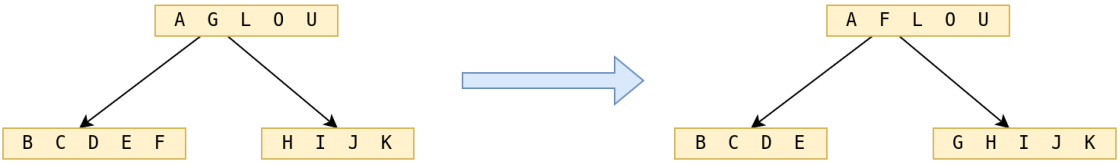

Figura 2.8: Operação de rotação à direita. O nó `HIJK` recebeu de seu pai `AGLOU` a chave `G`; seu pai, por sua vez, recebeu do nó `BCDEF` a chave `F`.

Continuando o estudo das funções auxiliares, o procedimento `CONCATENATE-LEFT` abaixo recebe páginas x, y, z e um índice i tal que `x.filho[i]` $= y$, `x.filho[i+1]` $= z$ e `y.n` $=$ `z.n` $= t-1$. Tal função move `x.chave[i]` e `x.valor[i]` para y, movendo para y também todas as chaves e valores presentes em z, além de ajustar os ponteiros para as páginas filhas, caso seja necessário.

Algorithm 9

```
1:  procedure CONCATENATE-LEFT(x, y, z, i)
2:      y.chave[t] ← x.chave[i]
3:      y.valor[t] ← x.valor[i]
4:      for j = 1 to t - 1 do
5:          y.chave[j + t] ← z.chave[j]
6:          y.valor[j + t] ← z.valor[j]
7:      if not z.folha then
8:          for j = 1 to t do
9:              y.filho[j + t] ← z.filho[j]
10:     for j = i to x.n - 1 do
11:         x.chave[j] ← x.chave[j + 1]
12:         x.valor[j] ← x.valor[j + 1]
13:     for j = i + 1 to x.n do
14:         x.filho[j] ← x.filho[j + 1]
15:     x.n ← x.n - 1
16:     y.n ← 2t - 1
17:     DISK-WRITE(x)
18:     DISK-WRITE(y)
19:     DISK-DELETE(z)
```

As linhas 2-3 do procedimento `CONCATENATE-LEFT` inserem em y a chave `x.chave[`i`]` e o valor `x.valor[`i`]`, e o loop nas linhas 4-6 insere em y todas as chaves e valores presentes em z. Caso z não seja do tipo folha, então o loop nas linhas 7-9 move para y os ponteiros para os filhos de z adequadamente. Em seguida, o loop nas linhas 10-12 acerta a posição das chaves e valores em x, após mover `x.chave[`i`]` de x para y, e o loop das linhas 13-14 atualiza a posição dos ponteiros para os filhos. Por fim, as linhas 15-16 corrigem o número de chaves presentes nas páginas x e y, e as linhas 17-19 gravam todas as alterações em disco, removendo a página z que não será mais usada da memória secundária usando a função `DISK-DELETE` (abstraída da mesma maneira que `DISK-READ` e `DISK-WRITE`).

Os loops das linhas 4-6 e 8-9 são executados na ordem de t vezes, enquanto os loops das linhas 10-12 e 13-14 são executados na ordem de `x.n` vezes. Como `x.n` é limitado por $2t - 1$, e t é definido de antemão, temos que a complexidade de tempo do algoritmo `CONCATENATE-LEFT` é $\Theta(1)$.

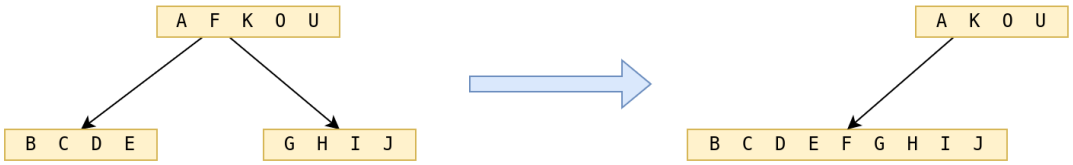

Figura 2.9: Operação de concatenação à esquerda. O nó `BCDE` recebeu a chave `F`, pertencente ao seu pai `AFKOU`, bem como todas as chaves de seu irmão direito `GHIJ`.

Também temos a função `CONCATENATE-RIGHT`, que possui comportamento similar ao observado em `ROTATE-LEFT`. A função `ROTATE-RIGHT` recebe páginas x, y, z e um índice i tal que `x.filho[`i`]` $= y$ e `x.filho[`$i+1$`]` $= z$. Além disso, também devemos ter `y.n` $= $ `z.n` $= t - 1$. O objetivo de tal função é concatenar as páginas y e z à direita, ou seja, gravar em z a chave `x.filho[`i`]`, o valor `x.valor[`i`]` e todas as chaves e valores de y, acertando os ponteiros para os nós filhos, conforme necessário.

Algorithm 10

```
1:  procedure CONCATENATE-RIGHT(x, y, z, i)
2:      for j = 1 to t - 1 do
3:          z.chave[j + t] ← z.chave[j]
4:          z.valor[j + t] ← z.valor[j]
5:      for j = 1 to t - 1 do
6:          z.chave[j] ← y.chave[j]
7:          z.valor[j] ← y.valor[j]
8:      z.chave[t] ← x.chave[i]
9:      z.valor[t] ← x.valor[i]
10:     if not z.folha then
11:         for j = 1 to t do
12:             z.filho[j + t] ← z.filho[j]
13:         for j = 1 to t do
14:             z.filho[j] ← y.filho[j]
15:     for j = i to x.n - 1 do
16:         x.chave[j] ← x.chave[j + 1]
17:         x.valor[j] ← x.valor[j + 1]
18:     for j = i to x.n do
19:         x.filho[j] ← x.filho[j + 1]
20:     x.n ← x.n - 1
21:     z.n ← 2t - 1
22:     DISK-WRITE(x)
23:     DISK-WRITE(z)
24:     DISK-DELETE(y)
```

O loop nas linhas 2-4 desloca em t unidades as chaves e valores já existentes na página z. Em seguida, o loop nas linhas 5-7 insere em z as $t-1$ chaves e os $t-1$ valores presentes em y. As linhas 8-9 inserem em z a chave `x.chave[`i`]` e o valor `x.valor[`i`]`, e caso o nó z não seja do tipo folha, os loops das linhas 11-12 e 13-14 acertam os ponteiros para os nós filhos. Adiante, o loop das linhas 15-17 atualiza as chaves e valores em x, considerando-se a remoção de `x.chave[`i`]`, e as linhas 18-19 ajustam os ponteiros para os filhos de x. As linhas 20-21 acertam os valores de `x.n` e `z.n` após esta sequência de operações, e as linhas 22-24 gravam todas as alterações na memória secundária, incluindo a remoção do nó y que não será mais usado.

Todos os loops presentes nesta função são executados menos de $2t$ vezes. Assim sendo, tendo em vista que t é uma constante definida de antemão, temos que a complexidade de tempo deste algoritmo é $\Theta(1)$.

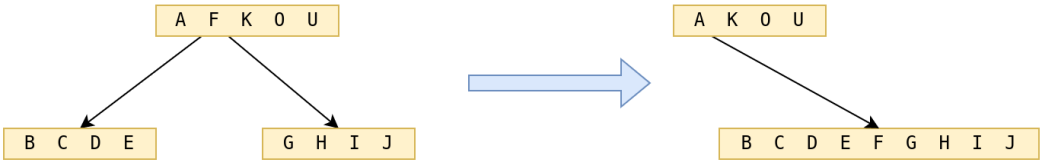

Figura 2.10: Operação de concatenação à direita. O nó `GHIJ` recebeu de seu pai `AFKOU` a chave `F`, bem como todas as chaves de seu irmão esquerdo `BCDE`.

Prosseguindo, temos a função `DELETE-MINIMUM` abaixo, que recebe um nó interno x, cujo número de chaves `x.n` é pelo menos t, e remove da subárvore com raiz em x sua chave mínima, retornando o par (chave, valor) correspondente.

Algorithm 11

```
1:  procedure DELETE-MINIMUM(x)
2:      y ← x.filho[1]
3:      z ← x.filho[2]
4:      DISK-READ(y)
5:      if y.folha then
6:          if y.n > t - 1 then
7:              min ← (y.chave[1], y.valor[1])
8:              for j = 1 to y.n - 1 do
9:                  y.chave[j] ← y.chave[j + 1]
10:                 y.valor[j] ← y.valor[j + 1]
11:             y.n ← y.n - 1
12:             DISK-WRITE(y)
13:             return min
14:         else
15:             DISK-READ(z)
16:             if z.n > t - 1 then
17:                 min ← (y.chave[1], y.valor[1])
18:                 for j = 1 to t - 1 do
19:                     y.chave[j] ← y.chave[j + 1]
20:                     y.valor[j] ← y.valor[j + 1]
21:                 y.n ← y.n - 1
22:                 ROTATE-LEFT(x, y, z, 1)
23:                 return min
24:             else
25:                 CONCATENATE-LEFT(x, y, z, 1)
26:                 min ← (y.chave[1], y.valor[1])
27:                 for j = 1 to 2t - 2 do
28:                     y.chave[j] ← y.chave[j + 1]
29:                     y.valor[j] ← y.valor[j + 1]
30:                 y.n ← 2t - 2
31:                 DISK-WRITE(y)
32:                 return min
33:     else
34:         if y.n > t - 1 then
35:             return DELETE-MINIMUM(y)
36:         else
37:             DISK-READ(z)
38:             if z.n > t - 1 then
39:                 ROTATE-LEFT(x, y, z, 1)
40:                 return DELETE-MINIMUM(y)
41:             else
42:                 CONCATENATE-LEFT(x, y, z, 1)
43:                 return DELETE-MINIMUM(y)
```

As linhas 2-3 atribuem a y e z os ponteiros para os filhos `x.filho[1]` e `x.filho[2]`, respectivamente, com o objetivo de melhorar a legibilidade no decorrer do código. Como x é um nó interno, temos que o mínimo da subárvore com raiz em x deve estar necessariamente em y. A depender do tipo de y (interno ou folha) e do número de chaves ali presentes, teremos vários casos a tratar:

- Caso y seja um nó folha, e o número de chaves em y seja maior do que $t - 1$ (linha 6), então as linhas 7-13 removem o mínimo de y e retornam o par (chave, valor) associado.
- Caso y seja um nó folha, possua $t - 1$ chaves e seu irmão z possua mais do que $t - 1$ chaves (linha 16), então, ao final da remoção (linhas 17-21), deverá ser realizada uma operação do tipo `ROTATE-LEFT` (linha 22), para que y continue a conter $t - 1$ chaves e preserve as propriedades de uma árvore B. O par (chave, valor) é retornado na linha 23.
- Caso y seja um nó folha, possua $t - 1$ chaves e seu irmão z também possua $t - 1$ chaves (linha 24),

então deverá ser realizada uma operação do tipo `CONCATENATE-LEFT` (linha 25), seguida pela remoção do mínimo em y (linhas 26-30). As alterações são gravadas em disco (linha 31), e o par (chave, valor) é retornado na linha 32.

- Caso y seja um nó interno e possua pelo menos t chaves (linha 34), então `DELETE-MINIMUM` é chamado recursivamente no nó y (linha 35).
- Caso y seja um nó interno, possua $t - 1$ chaves e seu irmão z possua pelo menos t chaves (linha 38), então primeiramente deverá ser realizada uma operação do tipo `ROTATE-LEFT` (linha 39), para garantir que y possua t chaves ao final do procedimento. Em seguida, podemos chamar recursivamente a função `DELETE-MINIMUM` no nó y (linha 40).
- Caso y seja um nó interno, possua $t-1$ chaves e seu irmão z também possua $t-1$ chaves (linha 41), então primeiramente deverá ser realizada uma operação do tipo `CONCATENATE-LEFT` (linha 42). Ao final do procedimento, o nó y conterá $2t - 1$ chaves. Em seguida chamamos recursivamente a função `DELETE-MINIMUM` em y (linha 43).

Como já vimos, os procedimentos `ROTATE-LEFT` e `CONCATENATE-LEFT` são executados em tempo $\Theta(1)$. Além disso, cada um dos loops de `DELETE-MINIMUM` é executado um número de vezes da ordem de t (que é constante; definido de antemão). Portanto, cada chamada recursiva a `DELETE-MINIMUM` é executada em tempo constante. Como o mínimo é removido de um nó folha, temos de descer por toda a árvore, que possui altura h. Logo, a complexidade de tempo de `DELETE-MINIMUM` é $\Theta(h) = \Theta(\log_t n) = \Theta(\lg n)$.

O procedimento `DELETE-MAXIMUM`, descrito abaixo, recebe um nó interno x com pelo menos t chaves, remove a chave máxima da subárvore com raiz em x e retorna o par (chave, valor) associado.

Algorithm 12

```
1:  procedure DELETE-MAXIMUM(x)
2:      y ← x.filho[x.n]
3:      z ← x.filho[x.n + 1]
4:      DISK-READ(z)
5:      if z.folha then
6:          if z.n > t - 1 then
7:              z.n ← z.n - 1
8:              DISK-WRITE(z)
9:              return (z.chave[z.n + 1], z.valor[z.n + 1])
10:         else
11:             DISK-READ(y)
12:             if y.n > t - 1 then
13:                 max ← (z.chave[z.n], z.valor[z.n])
14:                 z.n ← z.n - 1
15:                 ROTATE-RIGHT(x, y, z, x.n)
16:                 return max
17:             else
18:                 CONCATENATE-LEFT(x, y, z, x.n)
19:                 y.n ← y.n - 1
20:                 DISK-WRITE(y)
21:                 return (y.chave[y.n + 1], y.valor[y.n + 1])
22:     else
23:         if z.n > t - 1 then
24:             return DELETE-MAXIMUM(z)
25:         else
26:             DISK-READ(y)
27:             if y.n > t - 1 then
28:                 ROTATE-RIGHT(x, y, z, x.n)
29:                 return DELETE-MAXIMUM(z)
30:             else
31:                 CONCATENATE-LEFT(x, y, z, x.n)
32:                 return DELETE-MAXIMUM(y)
```

As linhas 2-3 atribuem a y e z os ponteiros para os filhos `x.filho[x.n]` e `x.filho[x.n + 1]`, respectivamente, a fim de melhorar a legibilidade no código. Como x é um nó interno, sabemos que o máximo da subárvore com raiz em x deve estar necessariamente em z, pelas propriedades de uma árvore B. A depender do tipo de z (interno ou folha) e do número de chaves ali presentes, teremos vários casos a tratar:

- Caso o nó z seja do tipo folha e possua pelo menos t chaves (linha 6), então simplesmente removemos o máximo de z e o retornamos (linhas 7-9).
- Caso o nó z seja do tipo folha, possua $t - 1$ chaves e seu irmão y possua pelo menos t chaves (linha 12), então removemos o máximo de z (linhas 13-14), em seguida realizamos uma operação `ROTATE-RIGHT` (linha 15) para deixar z com $t - 1$ chaves. O par (chave, valor) removido de z é retornado na linha 16.
- Caso o nó z seja do tipo folha, possua $t - 1$ chaves e seu irmão y também possua $t - 1$ chaves (linha 17), então primeiramente é realizada uma operação `CONCATENATE-LEFT` (linha 18). Em seguida, o par (chave, valor) associado ao máximo é removido de y e retornado (linhas 20-21).
- Caso o nó z seja interno e possua pelo menos t chaves (linha 23), então `DELETE-MAXIMUM` é chamado recursivamente em z (linha 24).
- Caso o nó z seja interno, possua $t - 1$ chaves e seu irmão y possua pelo menos t chaves (linha 27), então uma operação do tipo `ROTATE-RIGHT` é realizada (linha 28). Em seguida, invocamos recursivamente a função `DELETE-MAXIMUM` em z (linha 29). A operação `ROTATE-RIGHT` é necessária para garantirmos a hipótese de chamar a função `DELETE-MAXIMUM` sobre um nó com pelo menos t filhos.

- Caso o nó z seja interno e tanto z quanto y possuam $t-1$ chaves (linha 30), então é realizada uma operação `CONCATENATE-LEFT` (linha 31), em seguida continuamos com a recursão, invocando a função `DELETE-MAXIMUM` sobre y (linha 32), que a esta altura possui $2t-1$ chaves.

Como cada chamada recursiva a `DELETE-MAXIMUM` é executada em tempo $\Theta(1)$, e precisamos descer até uma folha para recuperar a chave máxima, temos que a complexidade de tempo de `DELETE-MAXIMUM` sobre uma árvore de altura h é $\Theta(h)$.

De posse das funções anteriores, estamos prontos para estudar a função `B-TREE-DELETE-ON-NODE` abaixo, que recebe uma página x, uma chave k e a remove da subárvore com raiz em x, caso tal chave esteja presente, removendo também seu valor associado. A página x, caso seja um nó diferente da raiz, deverá necessariamente possuir pelo menos t chaves, a fim de preservar as propriedades da árvore B ao final da remoção. Por outro lado, se x for a raiz da árvore B, então necessariamente pelo menos uma das duas situações deverá ocorrer:

- x possui ao menos duas chaves.
- x possui ao menos um filho com no mínimo t chaves.

Se pelo menos uma destas situações ocorrer, então garantimos que chamadas futuras a funções do tipo `CONCATENATE` ou `ROTATE` envolvendo o nó raiz x não irão produzir uma árvore B contendo uma raiz x degenerada, ou seja, sem chaves e possuindo apenas um filho, ou com um filho contendo $t-2$ chaves. O caso problemático, em que a raiz x possui apenas uma chave e seus filhos possuem ambos $t-1$ chaves, será tratado posteriormente.

Algorithm 13

```
1: procedure B-TREE-DELETE-ON-NODE(x, k)
2:     i ← x.n
3:     while i > 0 and x.chave[i] > k do
4:         i ← i - 1
5:     if i > 0 and x.chave[i] = k then
6:         (chave, valor) ← (x.chave[i], x.valor[i])
7:         if x.folha then
8:             while i < x.n do
9:                 x.chave[i] ← x.chave[i + 1]
10:                x.valor[i] ← x.valor[i + 1]
11:            x.n ← x.n - 1
12:            return (chave, valor)
13:        else
14:            y ← x.filho[i]
15:            z ← x.filho[i + 1]
16:            DISK-READ(z)
17:            if z.n > t - 1 then
18:                if z.folha then
19:                    (x.chave[i], x.valor[i]) ← (z.chave[1], z.valor[1])
20:                    for j = 1 to z.n - 1 do
21:                        z.chave[j] ← z.chave[j + 1]
22:                        z.valor[j] ← z.valor[j + 1]
23:                    z.n ← z.n - 1
24:                else (x.chave[i], x.valor[i]) ← DELETE-MINIMUM(z)
25:                return (chave, valor)
26:            else
27:                DISK-READ(y)
28:                if y.n > t - 1 then
29:                    if y.folha then
30:                        (x.chave[i], x.valor[i]) ← (y.chave[y.n], y.valor[y.n])
31:                        y.n ← y.n - 1
32:                    else (x.chave[i], x.valor[i]) ← DELETE-MAXIMUM(y)
33:                    return (chave, valor)
34:                else
35:                    CONCATENATE-LEFT(x, y, z, i)
36:                    return B-TREE-DELETE-ON-NODE(y, k)
37:    else
38:        if x.folha then return NULL
39:        if i = 0 then
40:            i ← i + 1
41:            y ← x.filho[i]
42:            z ← x.filho[i + 1]
43:            DISK-READ(y)
44:            if y.n = t - 1 then
45:                DISK-READ(z)
46:                if z.n > t - 1 then ROTATE-LEFT(x, y, z, i)
47:                else CONCATENATE-LEFT(x, y, z, i)
48:            return B-TREE-DELETE-ON-NODE(y, k)
49:        else
50:            y ← x.filho[i]
51:            z ← x.filho[i + 1]
52:            DISK-READ(z)
53:            if z.n = t - 1 then
54:                DISK-READ(y)
55:                if y.n > t - 1 then ROTATE-RIGHT(x, y, z, i)
56:                else CONCATENATE-RIGHT(x, y, z, i)
57:            return B-TREE-DELETE-ON-NODE(z, k)
```

As linhas 2-4 procuram a posição adequada para prosseguir com a remoção. Após o loop da linha 3, uma das seguintes situações estará necessariamente acontecendo:

1. $i > 0$ e `x.chave[`i`]` $= k$. Se esta situação ocorrer, então devemos tratar os seguintes casos:

   - Se x for um nó folha (linha 7), então simplesmente removemos k e seu respectivo valor associado (linhas 8-11), já que, se $x \neq$ `T.raiz`, é garantido que x possua pelo menos t chaves. Assim sendo, ao final da remoção, as propriedades da árvore B continuarão sendo satisfeitas.
   - Se x for um nó interno (linha 13), então devemos substituir a chave k pela sua chave antecessora em `x.filho[`i`]` $= y$ ou pela sua chave sucessora em `x.filho[`$i+1$`]` $= z$. Se z possuir pelo menos t chaves, e for um nó folha (linha 18), então simplesmente removemos o sucessor de k em z e o substituimos em x, no lugar de k (linhas 19-23).
   - Se x for um nó interno, z possuir pelo menos t chaves e também for um nó interno (linha 24), então chamamos recursivamente o procedimento `DELETE-MINIMUM` sobre z, e substituimos k em x, bem como seu valor associado, pelo valor retornado por tal função.
   - Se x for um nó interno, z possuir exatamente $t-1$ chaves e seu irmão y possuir pelo menos t chaves e for do tipo folha (linha 29), então as linhas 30-31 removem o antecessor de k em y e substituem k em x pelo antecessor removido.
   - Se x for um nó interno, y também for um nó interno e possuir pelo menos t chaves (linha 32), então o procedimento `DELETE-MAXIMUM` será invocado em y, e o valor retornado substituirá k em x.
   - Se x for um nó interno, e se ambos y, z possuírem exatamente $t-1$ chaves (linha 34), então será realizada uma operação do tipo `CONCATENATE-LEFT` (linha 35) e o procedimento `B-TREE-DELETE-ON-NODE` será chamado recursivamente em y (linha 36), que a esta altura possuirá $2t-1$ chaves.

2. $i > 0$ e `x.chave[`i`]` $< k$. Caso x seja um nó folha, então o algoritmo retorna `NULL` (linha 38). Caso contrário, se x for um nó interno, então a chave procurada, se estiver em algum lugar, estará no filho z de x, pelas propriedades da árvore B.

   - Se z possui pelo menos t chaves, então o procedimento `B-TREE-DELETE-ON-NODE` é chamado recursivamente em z (linha 57).
   - Se z possui exatamente $t-1$ chaves e seu irmão y possui pelo menos t chaves (linha 55), então será realizada uma operação do tipo `ROTATE-LEFT`, a fim de garantir que z tenha pelo menos t chaves, antes de chamarmos a função `B-TREE-DELETE-ON-NODE` recursivamente.
   - Se y e z possuem ambos $t-1$ chaves (linha 56), uma operação do tipo `CONCATENATE-RIGHT` será realizada, e a recursão continuará em z.

3. $i = 0$. Caso x seja um nó folha, então o algoritmo retorna `NULL` (linha 38). Caso contrário, a recursão deverá continuar em $y =$ `x.filho[1]`, já que a chave k, se estiver presente em algum lugar, deverá estar necessariamente nesta subárvore.

   - Se y possui pelo menos t chaves, então o procedimento `B-TREE-DELETE-ON-NODE` é chamado recursivamente em y (linha 48).
   - Se y possui exatamente $t-1$ chaves e seu irmão $z =$ `x.filho[2]` possui pelo menos t chaves (linha 46), então será realizada uma operação do tipo `ROTATE-LEFT`, a fim de garantir que y possua pelo menos t chaves. Em seguida, o procedimento `B-TREE-DELETE-ON-NODE` é chamado recursivamente em y (linha 48).
   - Se y e z possuírem ambos $t-1$ chaves (linha 47), então será realizada uma operação do tipo `CONCATENATE-LEFT`. Em seguida, tendo em vista que o nó y agora possui $2t-1$ chaves, será chamado recursivamente o procedimento `B-TREE-DELETE-ON-NODE` em y (linha 48).

Cada chamada recursiva a `B-TREE-DELETE-ON-NODE` possui complexidade de tempo $\Theta(1)$, já que os loops *while* das linhas 3-4 e 8-10 e *for* das linhas 20-22 são executados não mais que $2t-1$ vezes. Também temos que uma chamada a `B-TREE-DELETE-ON-NODE` necessariamente irá descer

até um nó folha, que está na profundidade h, seja para remover k de um nó folha, seja para encontrar o sucessor ou predecessor de k em um nó interno. Por esse motivo, a complexidade de tempo de `B-TREE-DELETE-ON-NODE` é $\mathcal{O}(h) = \mathcal{O}(\lg n)$.

Finalmente, a função `B-TREE-DELETE` abaixo recebe como argumentos um ponteiro para uma árvore B T, uma chave k e a remove de T, caso a mesma esteja presente na árvore. O par (chave, valor) é retornado em caso de sucesso, e `NULL` caso contrário.

Algorithm 14

```
1:  procedure B-TREE-DELETE(T, k)
2:      x ← T.raiz
3:      DISK-READ(x)
4:      if x.n = 1 and not x.folha then
5:          y = x.filho[1]
6:          DISK-READ(y)
7:          if y.n = t - 1 then
8:              z = x.filho[2]
9:              DISK-READ(z)
10:             if z.n = t - 1 then
11:                 CONCATENATE-LEFT(x, y, z, 1)
12:                 T.raiz ← y
13:     return B-TREE-DELETE-ON-NODE(T.raiz, k)
```

As linhas 2-3 realizam a leitura da raiz de T do disco. As linhas 4-12 cuidam do caso particular mencionado antes, em que `T.raiz` é um nó folha, contém uma única chave e cada um de seus dois filhos possui $t - 1$ chaves. Se tal caso ocorrer, então é realizada uma operação `CONCATENATE-LEFT`, e a nova raiz de T passará a ser o primeiro filho de `T.raiz`.

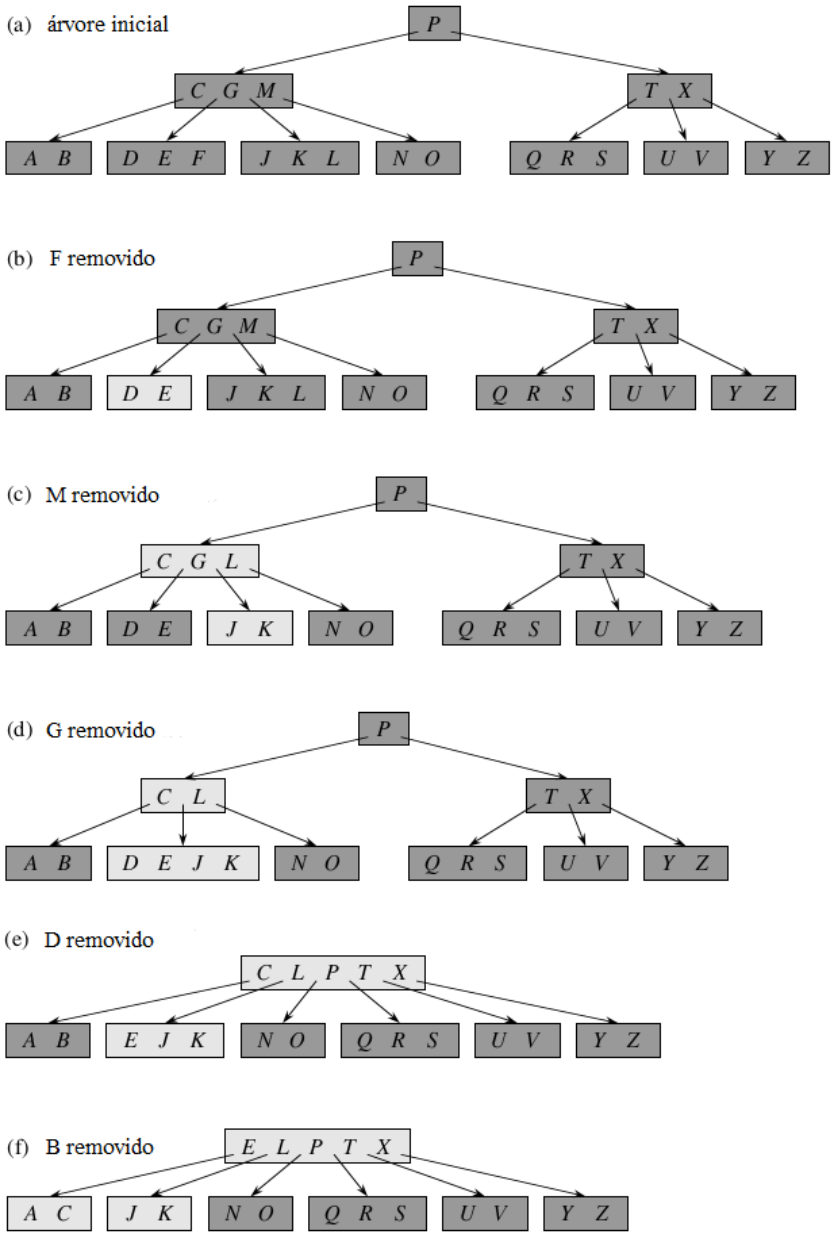

Figura 2.11: Remoção de chaves em uma árvore B, cujo grau mínimo é $t = 3$. Em **(a)**, a árvore B em seu estado inicial. Em **(b)**, a remoção da chave F em um nó folha DEF. Em **(c)**, a remoção da chave M em um nó interno GCM. A chave M foi substituída pela chave predecessora L. Em **(d)**, a remoção da chave G no nó interno GCM. Como ambos os filhos deste nó possuem $t - 1 = 2$ chaves, é realizada uma operação de concatenação à esquerda, seguida pela remoção de G deste novo nó. Em **(e)**, a remoção da chave D no nó folha DEJK. Como a raiz possui uma única chave, e seus dois filhos possuem ambos $t - 1 = 2$ chaves, é realizada uma operação de concatenação à esquerda, e a nova raiz passa a ser o resultado da concatenação. Em seguida, a chave D é removida normalmente do nó DEJK. Finalmente, em **(f)**, a remoção de B no nó folha AB. Como o número de chaves do nó AB é 2, primeiro é realizada uma rotação à esquerda, em seguida a chave B é removida. Fonte: CORMEN, Thomas et al. Introduction to Algorithms, 2nd edition, The MIT Press. (adaptado)

2.5 Usos práticos

Esta estrutura de dados é utilizada em sistemas gerenciadores de bancos de dados, bem como em sistemas de arquivos, como o NTFS, usado em sistemas baseados em Windows NT. Também é usada em outros sistemas de arquivos, como o HFS+ da Apple e Ext4 do Linux.

CAPÍTULO 3

ÁRVORES *SPLAY*

3.1 Introdução

Desenvolvida por Daniel Sleator e Robert Tarjan em 1985, as árvores *splay* são árvores binárias de busca autoajustáveis, com a propriedade de manter os elementos requisitados recentemente próximos à raiz, com o objetivo de reduzir o tempo de busca necessário para futuras requisições aos mesmos elementos. A base para o funcionamento de tal propriedade é uma operação chamada `SPLAY`, que traz o elemento inserido ou acessado recentemente até a raiz da árvore de busca, através do uso sucessivo de rotações, de tal modo que as propriedades de uma árvore binária de busca continuem sendo satisfeitas.

3.2 Rotações

A operação `SPLAY` traz um elemento à raiz da árvore *splay*, por meio de rotações sucessivas. Essas rotações podem ser de diversos tipos, e iremos entender cada uma delas a seguir.

3.2.1 Rotações simples do tipo ZIG e ZAG

Na rotação do tipo ZIG, se y possui um filho esquerdo x e é a raiz de uma subárvore, então fazemos apenas uma rotação em y para a direita, tornando y o filho direito de x. Na rotação do tipo ZAG, se x é a raiz de uma subárvore, e y é o seu filho direito, então fazemos apenas uma rotação em x para a esquerda, tornando x o filho esquerdo de y.

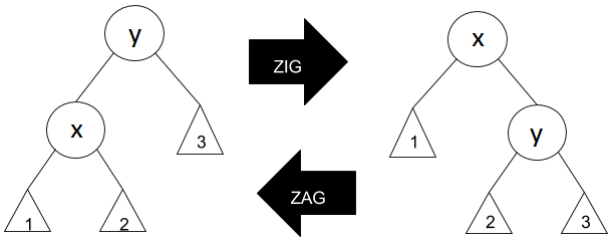

Figura 3.1: Exemplo de rotação simples do tipo ZIG (flecha da esquerda para a direita) e do tipo ZAG (flecha da direita para a esquerda). Fonte: Wikipédia

3.2.2 Rotações duplas do tipo ZIG-ZIG e ZAG-ZAG

Na rotação do tipo ZIG-ZIG, se y não é a raiz da árvore, e tanto y como x são filhos esquerdos de seus pais, então rotacionamos duas vezes para a direita: primeiro em z, depois em y. Na rotação do tipo ZAG-ZAG, se y não é a raiz da árvore, e tanto y como z são filhos direitos de seus pais, então rotacionamos duas vezes para a esquerda: primeiro em x, depois em y.

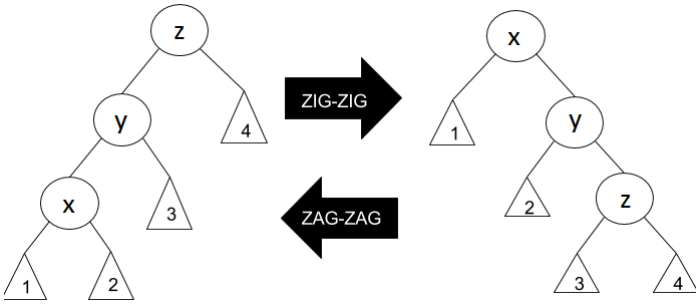

Figura 3.2: Exemplo de rotação dupla do tipo ZIG-ZIG (flecha da esquerda para a direita) e do tipo ZAG-ZAG (flecha da direita para a esquerda). Fonte: Wikipédia

3.2.3 Rotações duplas do tipo ZIG-ZAG e ZAG-ZIG

Na rotação do tipo ZIG-ZAG, se y não é a raiz da árvore, é filho direito de z e x é filho esquerdo de y, então rotacionamos y para a direita e, em seguida, z para a esquerda.

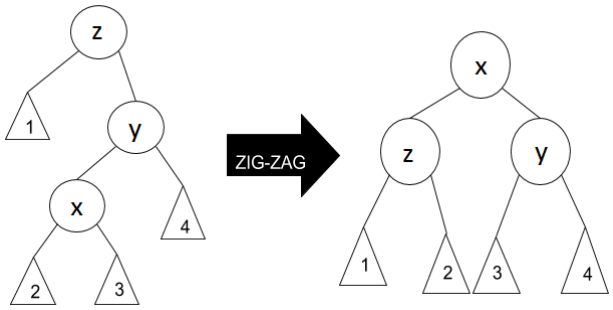

Figura 3.3: Exemplo de rotação dupla do tipo ZIG-ZAG. Fonte: Wikipédia

Na rotação do tipo ZAG-ZIG, se y não é a raiz da árvore, é filho esquerdo de z e x é filho direito de y, então rotacionamos y para a esquerda e, em seguida, z para a direita.

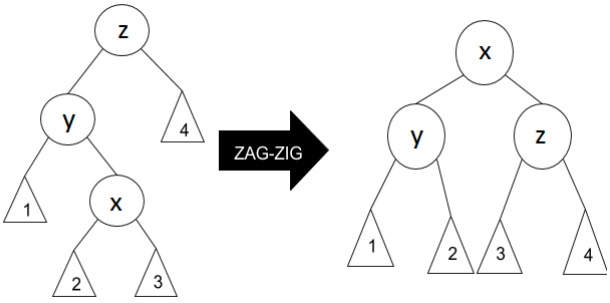

Figura 3.4: Exemplo de rotação dupla do tipo ZAG-ZIG. Fonte: Wikipédia

3.3 Operações básicas

3.3.1 Criando uma árvore *splay*

Como árvores *splay* são árvores binárias com a única propriedade adicional de manter o elemento acessado recentemente na raiz, para criar uma nova árvore *splay*, basta criar uma nova árvore de busca binária convencional. Esta operação está abstraída pela função `BTREE-CREATE`. Supõe-se que cada nó x da árvore de busca recém criada possua os campos `x.pai`, que aponta para o pai de x, `x.esquerdo`, que aponta para o filho esquerdo de x e `x.direito`, que aponta para o filho direito de x.

Algorithm 15

```
1: procedure SPLAY-CREATE( )
2:     return BTREE-CREATE()
```

3.3.2 Realizando uma operação *splay*

Antes de estudarmos o algoritmo que implementa a operação *splay*, precisamos entender dois algoritmos auxiliares, que realizam rotações à esquerda e à direita, em relação a determinado nó.

O algoritmo `ZIG` abaixo recebe como parâmetros uma árvore *splay* T, um nó x e rotaciona x à direita. O algoritmo assume que x possui o filho esquerdo. A ideia do algoritmo é fazer x descer um nível na árvore, pelo caminho à direita, e colocar o filho esquerdo y de x em seu lugar.

Algorithm 16

```
1: procedure ZIG(T, x)
2:     y ← x.esquerdo
3:     x.esquerdo ← y.direito
4:     if y.direito ≠ NULL then
5:         y.direito.pai ← x
6:     y.pai ← x.pai
7:     if x.pai = NULL then
8:         T.raiz ← y
9:     else
10:        if x = x.pai.esquerdo then
11:            x.pai.esquerdo ← y
12:        else
13:            x.pai.direito ← y
14:    y.direito ← x
15:    x.pai ← y
```

A linha 2 atribui a y o filho esquerdo de x (ele existe, por hipótese). Em seguida, as linhas 3-5 substituem y, o filho esquerdo de x, pelo filho direito de y. A linha 6 define o pai de y como sendo o pai de x. Caso o nó x não tenha pai (linha 7), então x inicialmente era a raiz de T, e tal lugar passará a ser ocupado por y (linha 8). Caso contrário, as linhas 10-13 substituem x por y adequadamente, alterando o nó `x.pai`. Por fim, as linhas 14-15 definem x como filho direito de y.

O algoritmo `ZAG` abaixo recebe como parâmetros uma árvore *splay* T, um nó x e rotaciona x à esquerda. O algoritmo assume que x possui o filho direito. Basicamente, o funcionamento do algoritmo é o seguinte: x desce um nível na árvore T, pelo caminho à esquerda, enquanto o filho direito de x, denotado por y, ocupa o seu lugar. Os ponteiros para os filhos de x e y são alterados conforme necessário.

Algorithm 17

```
1:  procedure ZAG(T, x)
2:      y ← x.direito
3:      x.direito ← y.esquerdo
4:      if y.esquerdo ≠ NULL then
5:          y.esquerdo.pai ← x
6:      y.pai ← x.pai
7:      if x.pai = NULL then
8:          T.raiz ← y
9:      else
10:         if x = x.pai.esquerdo then
11:             x.pai.esquerdo ← y
12:         else
13:             x.pai.direito ← y
14:     y.esquerdo ← x
15:     x.pai ← y
```

A linha 2 atribui a y o filho direito de x (ele existe, por hipótese). Em seguida, as linhas 3-5 substituem y, o filho direito de x, pelo filho esquerdo de y. A linha 6 define o pai de y como sendo o pai de x. Caso o nó x não tenha pai (linha 7), então x inicialmente era a raiz de T, e tal lugar passará a ser ocupado por y (linha 8). Caso contrário, as linhas 10-13 substituem x por y adequadamente, alterando o nó `x.pai`. Por fim, as linhas 14-15 definem x como filho esquerdo de y.

Finalmente, temos o algoritmo `SPLAY` abaixo, que recebe uma árvore *splay* T, um nó x e realiza uma operação *splay* sobre o nó x, ou seja, traz x até a raiz de T.

Algorithm 18

```
1:  procedure SPLAY(T, x)
2:      while x.pai ≠ NULL do
3:          if x.pai.pai = NULL then
4:              if x.pai.esquerdo = x then
5:                  ZIG(T, x.pai)
6:              else
7:                  ZAG(T, x.pai)
8:          else
9:              if x.pai.esquerdo = x and x.pai.pai.esquerdo = x.pai then
10:                 ZIG(T, x.pai.pai)
11:                 ZIG(T, x.pai)
12:             else
13:                 if x.pai.direito = x and x.pai.pai.direito = x.pai then
14:                     ZAG(T, x.pai.pai)
15:                     ZAG(T, x.pai)
16:                 else
17:                     if x.pai.esquerdo = x and x.pai.pai.direito = x.pai then
18:                         ZIG(T, x.pai)
19:                         ZAG(T, x.pai)
20:                     else
21:                         ZAG(T, x.pai)
22:                         ZIG(T, x.pai)
```

Enquanto `x.pai` $\neq$ `NULL`, o algoritmo trata os seguintes casos:

- Se x é filho esquerdo da raiz (linha 4), então é realizada uma rotação do tipo ZIG (linha 5).
- Se x é filho direito da raiz (linha 6), então é realizada uma rotação do tipo ZAG (linha 7)

- Se x não é filho da raiz, é filho esquerdo de `x.pai` e seu pai também é filho esquerdo de `x.pai.pai` (linha 9), então é realizada uma rotação do tipo ZIG-ZIG (linhas 10-11).
- Se x não é filho da raiz, é filho direito de `x.pai` e seu pai também é filho direito de `x.pai.pai` (linha 13), então é realizada uma rotação do tipo ZAG-ZAG (linhas 14-15).
- Se x não é filho da raiz, é filho esquerdo de `x.pai` e seu pai é filho direito de `x.pai.pai` (linha 17), então é realizada uma rotação do tipo ZIG-ZAG (linhas 18-19).
- Se x não é filho da raiz, é filho direito de `x.pai` e seu pai é filho esquerdo de `x.pai.pai` (linha 20), então é realizada uma rotação do tipo ZAG-ZIG (linhas 21-22).

A alteração do ponteiro `x.pai` é realizada pelas funções `ZIG` e `ZAG`, de modo que, a cada chamada a tais funções, o nó x sobe um nível na árvore binária de busca. O algoritmo termina quando `x.pai` $=$ `NULL`, ou seja, quando $x =$ `T.raiz`, como queremos.

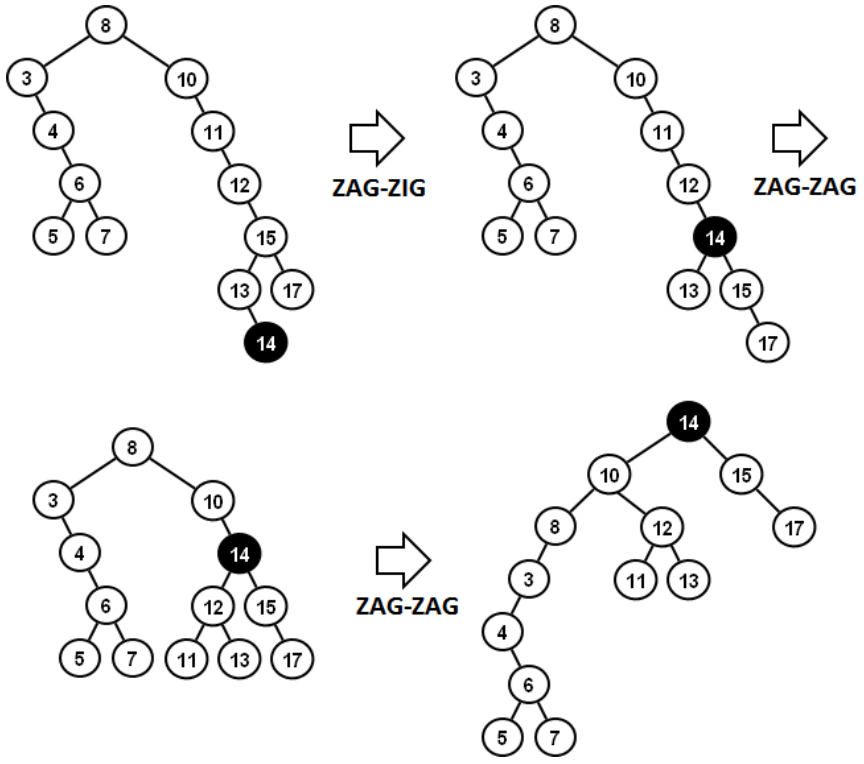

Figura 3.5: Exemplo de operação *splay* sobre o nó 14.
Fonte: https://github.com/nate-browne/CSE12_Redesign/tree/master/images

3.3.3 Pesquisando uma chave

O algoritmo `SEARCH` recebe como argumentos uma árvore binária de busca T, uma chave k e retorna o nó associado a k caso a busca seja um sucesso, e `NULL` caso contrário.

Algorithm 19

```
1: procedure SEARCH(T, k)
2:     node ← BTREE-SEARCH(T, k)
3:     if node ≠ NULL then
4:         SPLAY(T, node)
5:     return node
```

A busca de um item em uma árvore de busca binária está abstraída pela função `BTREE-SEARCH` (linha 2). Espera-se que tal função retorne o nó que contém k como chave em caso de sucesso, e `NULL` caso

contrário. Se a busca foi um sucesso (linha 3), então é realizada uma operação `SPLAY` no nó encontrado, trazendo-o à raiz da árvore T. Por fim, o resultado da busca é retornado (linha 5).

3.3.4 Inserindo uma chave

O algoritmo `INSERT` recebe uma árvore binária de busca T, uma chave k e um valor v associado à chave k, e os insere em T. A inserção é abstraída pela função `BTREE-INSERT` (linha 2). Supõe-se que tal função retorna o nó associado à chave k após a inserção. Se a chave k já pertencer à árvore T, então seu valor é atualizado para v. Ao final, uma operação `SPLAY` é realizada no nó que contém k (linha 2).

Algorithm 20

```
1: procedure INSERT(T, k, v)
2:     node ← BTREE-INSERT(T, k, v)
3:     SPLAY(T, node)
```

3.3.5 Removendo uma chave

O algoritmo `DELETE` recebe uma árvore de busca binária T, uma chave k e remove k de T, caso $k \in T$. A remoção é abstraída pela função `BTREE-DELETE`, que recebe os mesmos argumentos e remove k de uma árvore binária de busca T. Supõe-se que tal função retorna o nó associado à chave k.

Algorithm 21

```
1: procedure DELETE(T, k)
2:     return BTREE-DELETE(T, k)
```

3.4 Análise amortizada

Realizaremos a análise amortizada sobre as operações elementares usando o método do potencial, visto em cursos de análise de algoritmos. Definimos:

- A função *size*: $s(x) =$ o número de nós na subárvore com raiz em x, incluindo x
- A função *rank*: $r(x) = \lg s(x)$
- A função potencial: $\Phi(T) = \sum_{x \in T} r(x)$

O valor de Φ tenderá a ser alto para árvores mal equilibradas, e baixo para árvores bem equilibradas. Para aplicar o método do potencial, primeiro calculamos a mudança no potencial $\Delta\Phi$, causada por uma operação de rotação. Vamos verificar cada caso separadamente, e iremos denotar por r' a função r após a operação de rotação.

Analisaremos a seguir as rotações do tipo ZIG, ZIG-ZIG e ZIG-ZAG. Por simetria, os resultados obtidos para ZAG, ZAG-ZAG e ZAG-ZIG serão os mesmos. Além disso, utilizaremos a mesma simbologia das figuras que estão na seção sobre rotações.

- **ZIG**

$$\begin{aligned}
\Delta\Phi &= (r'(x) + r'(y)) - (r(x) + r(y)) && \text{(apenas os nós } x \text{ e } y \text{ mudam o } \textit{rank}\text{)} \\
&= r'(y) - r(x) && \text{(pois } r'(x) = r(y)\text{)} \\
&\leqslant r'(x) - r(x) && \text{(pois } r'(y) < r'(x)\text{)}
\end{aligned}$$

- **ZIG-ZIG**

$$\begin{aligned}
\Delta\Phi &= (r'(x) + r'(y) + r'(z)) - (r(x) + r(y) + r(z)) && \text{(apenas os nós } x, y \text{ e } z \text{ mudam o } \textit{rank}\text{)} \\
&= r'(y) + r'(z) - r(x) - r(y) && \text{(pois } r'(x) = r(z)\text{)} \\
&\leqslant r'(x) + r'(z) - 2r(x) && \text{(pois } r(x) < r(y) \text{ e } r'(x) > r'(y)\text{)}
\end{aligned}$$

Usando-se o fato da função logaritmo ser côncava, temos:

$$\frac{r(x)+r'(z)}{2} = \frac{\lg(s(x))+\lg(s'(z))}{2} \leqslant \lg\left(\frac{s(x)+s'(z)}{2}\right) \overset{*}{\leqslant} \lg\left(\frac{s'(x)}{2}\right) = r'(x)-1$$

$(*$ pois $s(x)+s'(z) \leqslant s'(x))$

Portanto $r'(z) \leqslant 2r'(x) - r(x) - 2$, logo $\Delta\Phi \leqslant 3(r'(x) - r(x)) - 2$.

- **ZIG-ZAG**

$$\begin{aligned}\Delta\Phi &= (r'(x)+r'(y)+r'(z)) - (r(x)+r(y)+r(z)) \\ &\leqslant r'(y)+r'(z)-2r(x) && \text{(pois } r'(x)=r(z) \text{ e } r(x)<r(y)) \\ &\leqslant 3(r'(x)-r(x))-2 && \text{(pela concavidade da função logaritmo)}\end{aligned}$$

Se $\hat{c}_i$ denotar o custo amortizado da i-ésima operação que possui custo real c_i, então vale que $\hat{c}_i = c_i + \Delta\Phi$. Para operações do tipo ZIG ou ZAG, o custo real é $c_i = 1$, pois apenas uma rotação está envolvida. Logo, neste caso, $\hat{c}_i = r'(x) - r(x) + 1$.

Para os casos ZIG-ZIG, ZIG-ZAG, ZAG-ZAG ou ZAG-ZIG, o custo real por operação é $c_i = 2$, porque duas rotações estão envolvidas em cada operação destas. Logo o custo amortizado é $\hat{c}_i = 3(r'(x) - r(x))$.

Considerando-se que $r_i(x)$ é o *rank* de x no i-ésimo passo de rotação, o custo amortizado de uma operação *splay* será:

$$\begin{aligned}\Sigma\hat{c}_i &\leqslant 3\Sigma(r_{i+1}(x) - r_i(x)) + 1 \\ &= 3(r(\text{raiz}) - r(x)) + 1 && \text{(soma telescópica)} \\ &\leqslant 3\lg n + 1 \\ &= \mathcal{O}(\lg n)\end{aligned}$$

Além disso, como o custo real de acessar um nó x, a partir da raiz, é menor ou igual ao custo real de realizar a operação *splay* em x, temos que o custo amortizado para pesquisar um nó x é $\mathcal{O}(\lg n)$. Também temos que o custo amortizado para a operação de inserção é $\mathcal{O}(\lg n)$, pois tal operação envolve o custo de descer até um nó folha (amortizado $\mathcal{O}(\lg n)$) adicionado ao custo de realizar uma operação *splay* no novo nó recém inserido (amortizado $\mathcal{O}(\lg n)$). Por fim, o custo amortizado de uma operação de remoção é $\mathcal{O}(\lg n)$, pois a operação de remoção envolve achar um nó x para remoção (amortizado $\mathcal{O}(\lg n)$) e substituí-lo pelo seu antecessor ou sucessor (envolve encontrar tal nó, portanto custo amortizado $\mathcal{O}(\lg n)$).

Desse modo, podemos enunciar o seguinte teorema:

Teorema 3.4.1. *Qualquer operação básica sobre uma árvore splay que contém n nós possui tempo amortizado $\mathcal{O}(\lg n)$.*

□

3.5 Vantagens e desvantagens

Vantagens:

- O desempenho médio de uma árvore *splay* é tão eficiente quanto o desempenho médio de outras árvores.
- A implementação de árvores *splay* é fácil.
- O consumo de memória é pequeno, pois não é necessário armazenar nenhum tipo de contador nos nós da árvore *splay*.

Desvantagens:

- A altura de uma árvore *splay* pode ser linear. Por exemplo, ao acessarmos todos os elementos de uma árvore *splay* em ordem não decrescente.
- O formato da árvore *splay* muda com o passar do tempo, mesmo quando são usadas apenas para leitura dos dados. Este fato traz uma enorme dificuldade para usos em aplicações *multi-thread*.
- Quando o padrão de acesso aos dados é totalmente aleatório, a operação `splay` traz um custo significativo, em relação a outros tipos de árvores.

3.6 Usos práticos

Alguns usos práticos incluem:

- Implementação de *caches* e algoritmos para coleta de lixo.
- Presente em sistemas baseados no Windows NT, na gerência de memória virtual, redes e sistema de arquivos.
- Usada pelo compilador *gcc* e roteadores.

CAPÍTULO 4

HEAPS BINOMIAIS

4.1 Introdução

Desenvolvida pelo cientista da computação francês Jean Vuillemin em 1978, a estrutura de dados heap binomial é um tipo de heap semelhante ao heap binário em vários aspectos, possuindo, assim como o heap binário, tempo logarítmico para as operações de inserção, remoção e busca de mínimo ou máximo, a depender do tipo de heap. Entretanto, os heaps binomiais usam um outro tipo de estrutura em árvore, denominada árvore binomial. A diferença mais significativa entre um heap binário e um heap binomial é que este último possui a propriedade de unir rapidamente dois heaps em tempo logarítmico.

4.2 Árvores binomiais

Antes de apresentarmos a definição de heaps binomiais, precisamos entender o que é uma árvore binomial. É disso que trata esta seção.

4.2.1 Definição

Árvores binomiais são definidas do seguinte modo:

- Uma árvore binomial de ordem 0, denotada por B_0, é um único nó.
- Uma árvore binomial de ordem k, denotada por B_k, possui um nó raiz e seus filhos, da esquerda para a direita, são raízes de árvores binomiais B_{k-1}, B_{k-2}, ..., B_1, B_0.

Alternativamente, podemos entender uma árvore binomial B_k como sendo a união de duas árvores binomiais B_{k-1}, tal que a raiz de uma delas é o filho mais à esquerda da raiz da outra. Esta manipulação é a base da operação de união em um heap binomial, como iremos ver mais adiante.

Um nó de uma árvore binomial, denotado por x, possui os seguintes campos:

- `x.pai`: ponteiro para o nó pai de x; caso o nó x seja a raiz, então este campo valerá `NULL`.
- `x.filho`: ponteiro para o filho mais esquerdo de x; caso x não possua filhos, então este campo valerá `NULL`.
- `x.irmão`: ponteiro para o irmão imediatamente à direita de x; caso x não possua irmãos à direita, então este campo valerá `NULL`.
- `x.grau`: número de filhos que o nó x contém.
- `x.chave`: chave associada ao nó x.
- `x.valor`: valor associado à chave `x.chave`.

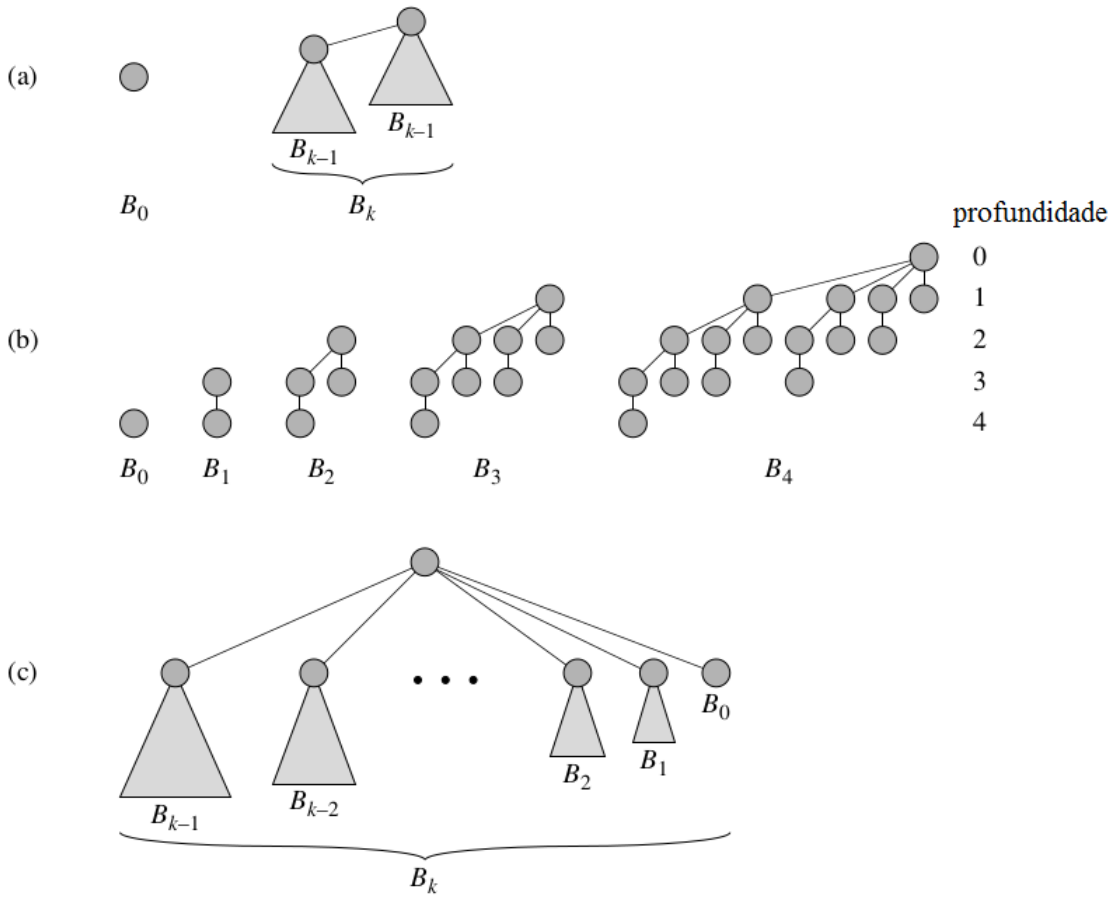

Figura 4.1: Representações de árvores binomiais. Em **(a)**, uma árvore binomial de ordem 0, e outra árvore binomial de ordem k, consistindo na junção de duas árvores binomiais de ordens $k-1$. Em **(b)**, mais algumas árvores binomiais, nas quais todos os seus nós estão explicitamente representados. Em **(c)**, uma árvore binomial de ordem k, consistindo em uma raiz que possui raízes de árvores binomiais de ordens $k-1, k-2, \ldots, 1, 0$ como filhos, nesta ordem, da direita para a esquerda. Fonte: CORMEN, Thomas et al. Introduction to Algorithms, 2nd edition, The MIT Press. (adaptado)

4.2.2 Propriedades

Uma árvore binomial B_k apresenta as seguintes propriedades:

1. Possui 2^k nós
2. Possui altura k
3. Possui exatamente $\binom{k}{i}$ nós na profundidade i, para $i \in \{0, 1, \ldots, k\}$
4. A raiz possui grau k, sendo o maior dentre todos os nós. Além disso, se numerarmos os filhos da raiz da esquerda para a direita, em ordem decrescente e a partir de $k-1$, temos que o i-ésimo filho é raiz de uma subárvore B_i.

Demonstração. É natural usar recursão em k para demonstrar as propriedades, já que as árvores binomiais são definidas recursivamente. A base da recursão de cada propriedade é a árvore B_0, e a validade das propriedades sobre B_0 é imediata. Para o passo indutivo, suponhamos que as propriedades sejam válidas para B_{k-1}.

1. Como B_k é a união de duas árvores B_{k-1}, temos que o número de nós de B_k é $2^{k-1} + 2^{k-1} = 2^k$.
2. Unindo-se duas árvores B_{k-1}, a altura aumenta em uma unidade. Como, por hipótese, a altura de uma árvore B_{k-1} é $k-1$, temos que a altura de B_k é k.

3. A linha i de B_k contém as linhas $i-1$ e i das duas árvores B_{k-1} que a compõem. Logo o número de elementos na linha i de B_k é, pela relação de Stifel:

$$\binom{k-1}{i-1} + \binom{k-1}{i} = \binom{k}{i}$$

4. A raiz de B_{k-1} possui, por hipótese, grau $k-1$. Assim sendo, a raiz de B_k possuirá grau k, porque tal árvore é a união de duas árvores B_{k-1}, com a raiz de uma sendo o filho mais à esquerda da outra. Além disso, por hipótese de indução, o i-ésimo filho de B_{k-1}, da esquerda para a direita, é raiz de uma subárvore B_i. Desse modo, ao acoplarmos duas árvores B_{k-1}, manteremos esta propriedade, pois os filhos da nova raiz, da esquerda para a direita, serão $B_{k-1}, B_{k-2}, \ldots, B_1, B_0$.

□

O teorema a seguir segue imediatamente das propriedades 1 e 4.

Teorema 4.2.1. *Se n for o número de nós de uma árvore binomial, então o grau máximo de qualquer nó é $\lg n$.*

□

Observação: o nome da estrutura é *heap binomial* por causa da propriedade 3.

4.3 Heaps binomiais

4.3.1 Definição

Um heap binomial H é definido como sendo uma coleção de árvores binomiais, satisfazendo as seguintes propriedades:

1. Toda árvore binomial $B_k \in H$ deve satisfazer a propriedade do heap mínimo, ou seja, para todo nó $x \in B_k$, se x não é a raiz de B_k, então `x.chave` > `x.pai.chave`
2. H possuirá no máximo uma árvore binomial de ordem k, para todo inteiro não negativo k.

Pela primeira propriedade, podemos entender que o valor mínimo de uma árvore binomial B_k se encontra na raiz de B_k. Pela segunda propriedade, podemos entender que, se H possui n nós no total, então existem no máximo $\lfloor \lg n \rfloor + 1$ árvores binomiais em H. Para ver o porquê disso, considere a representação binária de n, representada por

$$\langle b_{\lfloor \lg n \rfloor}, b_{\lfloor \lg n \rfloor - 1}, b_{\lfloor \lg n \rfloor - 2}, \ \ldots, b_1, b_0 \rangle \text{ tal que } \sum_{i=0}^{\lfloor \lg n \rfloor} b_i 2^i = n$$

Da representação binária de n, temos que uma árvore binomial $B_i \in H$ se e somente se $b_i = 1$.

4.3.2 Representação

Para representarmos um heap binomial H, usaremos uma lista ligada para unir os nós das árvores B_k, pertencentes a H, em ordem crescente. Esta lista eventualmente é mencionada como ***a lista de raízes***. Assim, se x_k é o nó raiz de uma árvore B_k pertencente a H, então seu campo x_k`.pai` valerá `NULL`, e seu campo x_k`.irmão` apontará para a raiz da próxima árvore percentente a H (se houver). Além disso, o heap binomial H conterá um campo `H.inicio`, que simplesmente apontará para a raiz da árvore binomial de menor ordem em H, ou valerá `NULL` caso o heap binomial H esteja vazio.

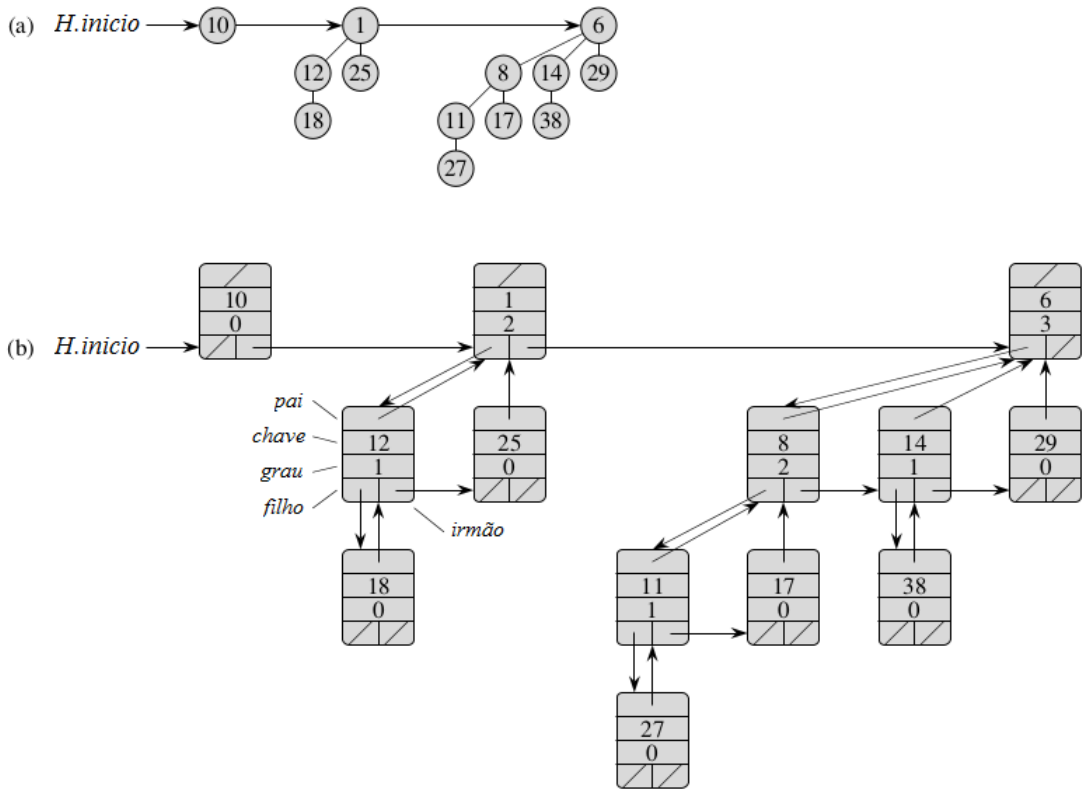

Figura 4.2: Representações gráficas de um heap binomial. Em **(a)**, um heap binomial, contendo três árvores binomiais. O início da lista de raízes é indicado pelo ponteiro `H.inicio`. As flechas indicam as ligações entre as raízes das árvores binomiais. Em **(b)**, o mesmo heap binomial, desta vez sendo explicitado todos os ponteiros associados a cada nó. Fonte: CORMEN, Thomas et al. Introduction to Algorithms, 2nd edition, The MIT Press. (adaptado)

4.4 Operações usuais

4.4.1 Criando um heap binomial

A função `BINOMIAL-HEAP-CREATE`, ao ser invocada, retorna um heap binomial vazio. A parte de alocação de memória foi abstraída pela função `ALLOCATE-MEMORY`, que retorna um ponteiro para alguma posição de memória contendo espaço suficiente para comportar o heap binomial H.

Algorithm 22

```
1: procedure BINOMIAL-HEAP-CREATE()
2:     H ← ALLOCATE-MEMORY()
3:     H.inicio ← NULL
4:     return H
```

4.4.2 Pesquisando a chave mínima

A função `BINOMIAL-HEAP-MINIMUM` recebe como argumento um heap binomial H e retorna o nó contendo a chave mínima de H.

Algorithm 23

```
1:  procedure BINOMIAL-HEAP-MINIMUM(H)
2:      y ← NULL
3:      x ← H.inicio
4:      min ← ∞
5:      while x ≠ NULL do
6:          if x.chave < min then
7:              min ← x.chave
8:              y ← x
9:          x ← x.irmão
10:     return y
```

O algoritmo realizará uma busca na lista de raízes, pois pela propriedade do heap mínimo, a raiz de cada árvore pertencente a H é o mínimo daquela árvore. Como tal lista possui tamanho máximo $\lfloor \lg n \rfloor + 1$, temos que seu tempo de execução é $\Theta(\lg n)$.

4.4.3 Unindo dois heaps binomiais

Antes de estudarmos o algoritmo que realiza a união entre dois heaps binomiais, precisamos entender o funcionamento de algumas funcões auxiliares.

O algoritmo `BINOMIAL-LINK` recebe dois nós y e z, que são raízes de árvores binomiais de ordem k, e torna y o filho mais à esquerda de z, ajustando os ponteiros adequadamente. Ao final do processo, z será raiz de uma árvore binomial de ordem $k + 1$. Tal algoritmo consome tempo $\Theta(1)$.

Algorithm 24

```
1:  procedure BINOMIAL-LINK(y, z)
2:      y.pai ← z
3:      y.irmão ← z.filho
4:      z.filho ← y
5:      z.grau ← z.grau + 1
```

O algoritmo `BINOMIAL-HEAP-MERGE` recebe dois heaps binomiais H_1 e H_2 e intercala a lista de raízes de ambos os heaps, em ordem não decrescente, usando o grau das árvores presentes nos heaps. Retorna uma única lista de raízes, contendo no máximo 2 árvores binomiais de ordem k, para todo inteiro não negativo k.

Algorithm 25

```
1:  procedure BINOMIAL-HEAP-MERGE(H_1, H_2)
2:      if H_1.inicio = NULL then return H_2.inicio
3:      if H_2.inicio = NULL then return H_1.inicio
4:      p ← H_1.inicio
5:      q ← H_2.inicio
6:      if p.grau < q.grau then
7:          cabeça ← r ← p
8:          p ← p.irmão
9:      else
10:         cabeça ← r ← q
11:         q ← q.irmão
12:     while p ≠ NULL and q ≠ NULL do
13:         if p.grau < q.grau then
14:             r.irmão ← p
15:             p ← p.irmão
16:         else
17:             r.irmão ← q
18:             q ← q.irmão
19:         r ← r.irmão
20:     if p ≠ NULL then r.irmão ← p
21:     else r.irmão ← q
22:     return cabeça
```

Caso algum dos heaps seja vazio (linhas 2-3), então o algoritmo simplesmente retorna o início da lista de raízes da outra árvore. Caso contrário, o algoritmo decide qual dos dois heaps contém no início de sua lista de raízes a árvore de menor grau (linhas 4-11). Neste ponto do algoritmo, os nós `cabeça` e r apontam para a raiz da árvore de menor grau em $H_1 \cup H_2$. Em seguida, o loop das linhas 12-19 adicionam à lista de raízes com início em `cabeça` o próximo nó raiz com menor grau dentre os restantes, sucessivamente, até que se tenha atingido o final da lista de raízes de algum heap binomial. Por fim, as linhas 20-21 adicionam as raízes restantes do outro heap (se houverem) à lista `cabeça` , e tal lista é retornada.

Se H_1 e H_2 contiverem n_1 e n_2 nós, respectivamente, então tais árvores terão no máximo $\lfloor \lg n_1 \rfloor + 1$ e $\lfloor \lg n_2 \rfloor + 1$ raízes em suas listas de raízes. Assim sendo, o consumo de tempo máximo do algoritmo `BINOMIAL-TREE-MERGE` é $\mathcal{O}(\lg n_1 + \lg n_2)$, pois o loop *while* consumirá no máximo $\lfloor \lg n_1 \rfloor + \lfloor \lg n_2 \rfloor + 2$ nós das listas de raízes.

Finalmente, estamos prontos para estudar o algoritmo de união. O algoritmo `BINOMIAL-HEAP-UNION` recebe como parâmetros dois heaps binomiais H_1 e H_2, e retorna um novo heap binomial $H = H_1 \cup H_2$. A representação dos dois heaps binomiais fornecidos como parâmetros são perdidas.

Algorithm 26

```
1:  procedure BINOMIAL-HEAP-UNION(H_1, H_2)
2:      H ← MAKE-BINOMIAL-HEAP()
3:      H.inicio ← BINOMIAL-HEAP-MERGE(H_1, H_2)
4:      if H.inicio = NULL then return H
5:      anterior-x ← NULL
6:      x ← H.inicio
7:      proximo-x ← x.irmao
8:      while proximo-x ≠ NULL do
9:          if x.grau ≠ proximo-x.grau ou
10:         (proximo-x.irmão ≠ NULL e proximo-x.irmão.grau = x.grau) then
11:             anterior-x ← x                                  ▷ Casos 1 e 2
12:             x ← proximo-x                                   ▷ Casos 1 e 2
13:         else
14:             if x.chave ⩽ proximo-x.chave then
15:                 x.irmão ← proximo-x.irmão                   ▷ Caso 3
16:                 BINOMIAL-LINK(proximo-x, x)                 ▷ Caso 3
17:             else
18:                 if anterior-x = NULL then H.inicio ← proximo-x   ▷ Caso 4
19:                 else anterior-x.irmão ← proximo-x           ▷ Caso 4
20:                 BINOMIAL-LINK(x, proximo-x)                 ▷ Caso 4
21:                 x ← proximo-x                               ▷ Caso 4
22:         proximo-x ← x.irmão
23:     return H
```

A linha 2 cria um novo heap binomial H, que conterá a união adequada entre os heaps binomiais H_1 e H_2. Em seguida, a lista de raízes dos dois heaps binomiais intercaladas é atribuída a `H.inicio` (linha 3). Se a lista de raízes intercalada for vazia (linha 4), então o algoritmo retorna o heap binomial H vazio. Caso contrário, sabemos que a lista de raízes intercalada possui pelo menos um elemento. A cada instante, os ponteiros `x`, `anterior-x` e `próximo-x` apontarão, respectivamente, para a raiz que está sendo examinada, a raiz anterior à raiz sendo examinada e a raiz sucessora à raiz sendo examinada. Inicialmente, tais ponteiros valerão, respectivamente: `NULL`, `H.inicio` e `H.inicio.irmão` (linhas 5-7). Durante a execução deste procedimento, pode acontecer da lista de raízes de H possuir, temporariamente, até três árvores binomiais de ordem k, para algum k. Vários casos deverão ser tratados durante as iterações do loop *while* na linha 8:

- Caso 1: Se o nó `x` sendo examinado e seu irmão `proximo-x` possuírem graus diferentes (linha 9), então simplesmente avançamos pela lista de raízes, atualizando os ponteiros (linhas 11-12 e 22).
- Caso 2: Se os nós `x`, `proximo-x` e `proximo-x.irmão` tiverem o mesmo grau (linha 10), então também avançamos pela lista de raízes, atualizando os ponteiros (linhas 11-12 e 22). Os nós `proximo-x` e `proximo-x.irmão` serão unidos na iteração seguinte.
- Caso 3: Se os nós `x` e `proximo-x` tiverem o mesmo grau, e `proximo-x` e `proximo-x.irmão` tiverem graus diferentes (linha 13), então uniremos os nós `x` e `proximo-x`. Se `x.chave` $\leqslant$ `proximo-x.chave`, então `proximo-x` se tornará o filho mais esquerdo de `x` (linhas 15-16).
- Caso 4: Caso contrário (linha 17), `x` se tornará o filho mais esquerdo de `proximo-x`. A linha 18 trata o caso em que `x` está no começo da lista de raízes, e a linha 19 trata o caso geral. A união das árvores é realizada na linha 20, e os ponteiros são atualizados nas linhas 21 e 22.

A complexidade de tempo de `BINOMIAL-HEAP-UNION` é $\mathcal{O}(\lg n)$, pois, supondo que os heaps H_1 e H_2 possuam respectivamente n_1 e n_2 nós, temos que o tempo de execução de `BINOMIAL-HEAP-MERGE` é proporcional a $\lfloor \lg n_1 \rfloor + \lfloor \lg n_2 \rfloor + 2$. Além disso, definindo-se $n := n_1 + n_2$, temos que

$$\lfloor \lg n_1 \rfloor + \lfloor \lg n_2 \rfloor + 2 \leqslant \lfloor \lg n \rfloor + \lfloor \lg n \rfloor + 2 = \mathcal{O}(\lg n)$$

e portanto o tempo de execução de `BINOMIAL-HEAP-MERGE` é $\mathcal{O}(\lg n)$. Além disso, o loop *while* do procedimento `BINOMIAL-HEAP-UNION` é executado não mais que $\lfloor \lg n_1 \rfloor + \lfloor \lg n_2 \rfloor + 2$ vezes, porque, a cada iteração, ou é realizada uma simples atualização de ponteiros, ou é executada uma operação de união entre árvores binomiais, que possui tempo de execução constante. Assim sendo, o consumo total de tempo do algoritmo `BINOMIAL-HEAP-UNION` é $\mathcal{O}(\lg n)$.

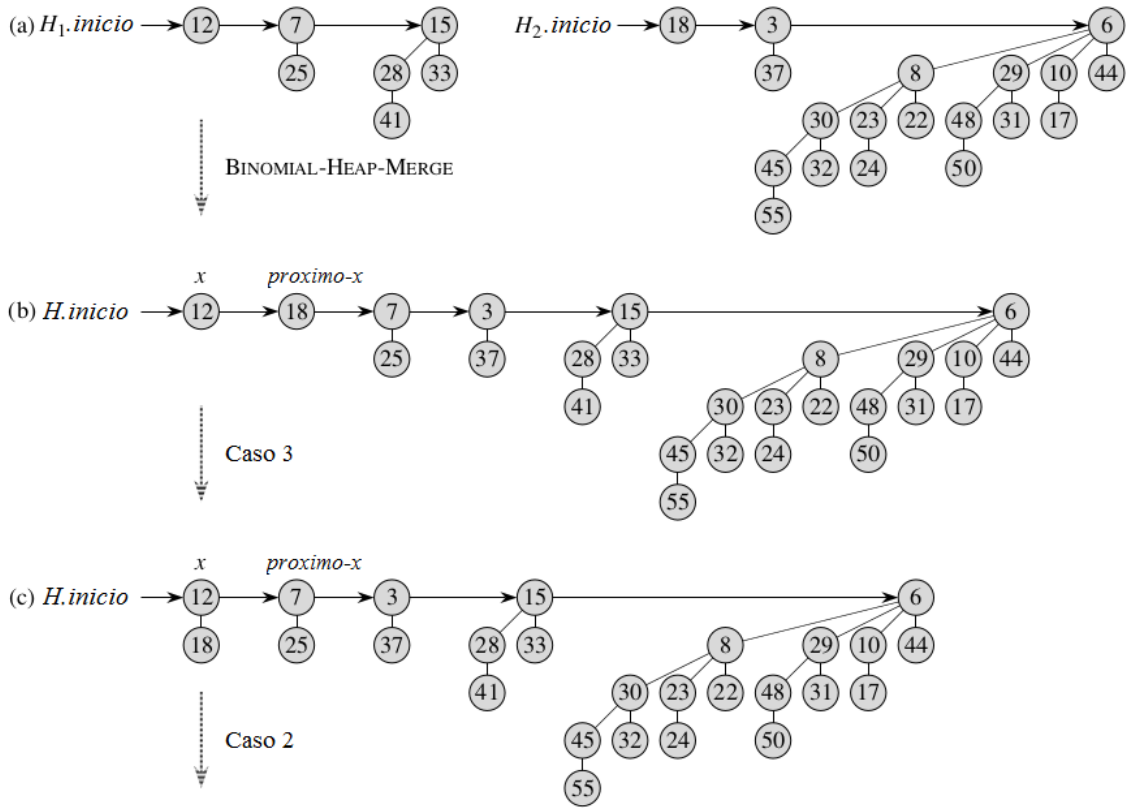

Figura 4.3:
Operação de união entre dois heaps H_1 e H_2. Em **(a)**, a representação dos dois heaps binomiais. Em **(b)**, o resultado de uma chamada a `BINOMIAL-HEAP-MERGE` sobre os dois heaps. Em **(c)**, as árvores com raízes 12 e 18 foram unidas, pois possuíam o mesmo grau. Fonte: CORMEN, Thomas et al. Introduction to Algorithms, 2nd edition, The MIT Press. (adaptado)

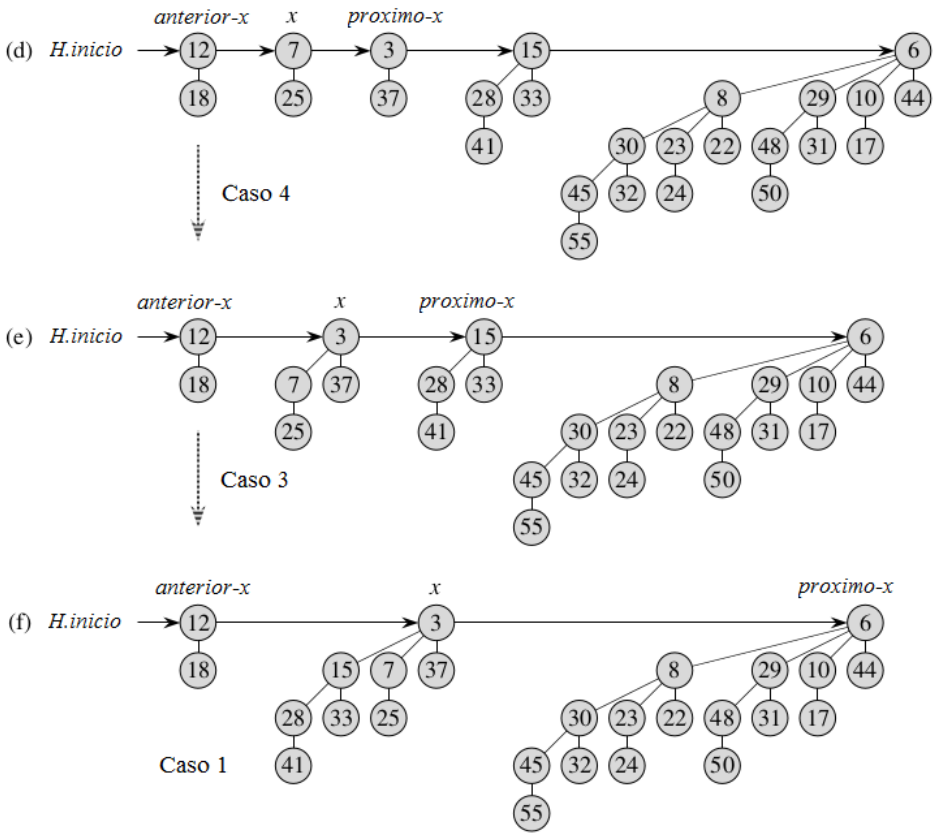

Figura 4.4: Em **(d)**, os ponteiros `x`, `anterior-x` e `proximo-x` foram simplesmente atualizados, pois existiam três raízes de mesmo grau, em sequência (caso 2). Em **(e)**, as árvores com raízes 7 e 3 foram unidas, pois a situação se enquadra no caso 4. Em **(f)**, as árvores com raízes 3 e 15 foram unidas, pois a situação se enquadra no caso 3. Fonte: CORMEN, Thomas et al. Introduction to Algorithms, 2nd edition, The MIT Press. (adaptado)

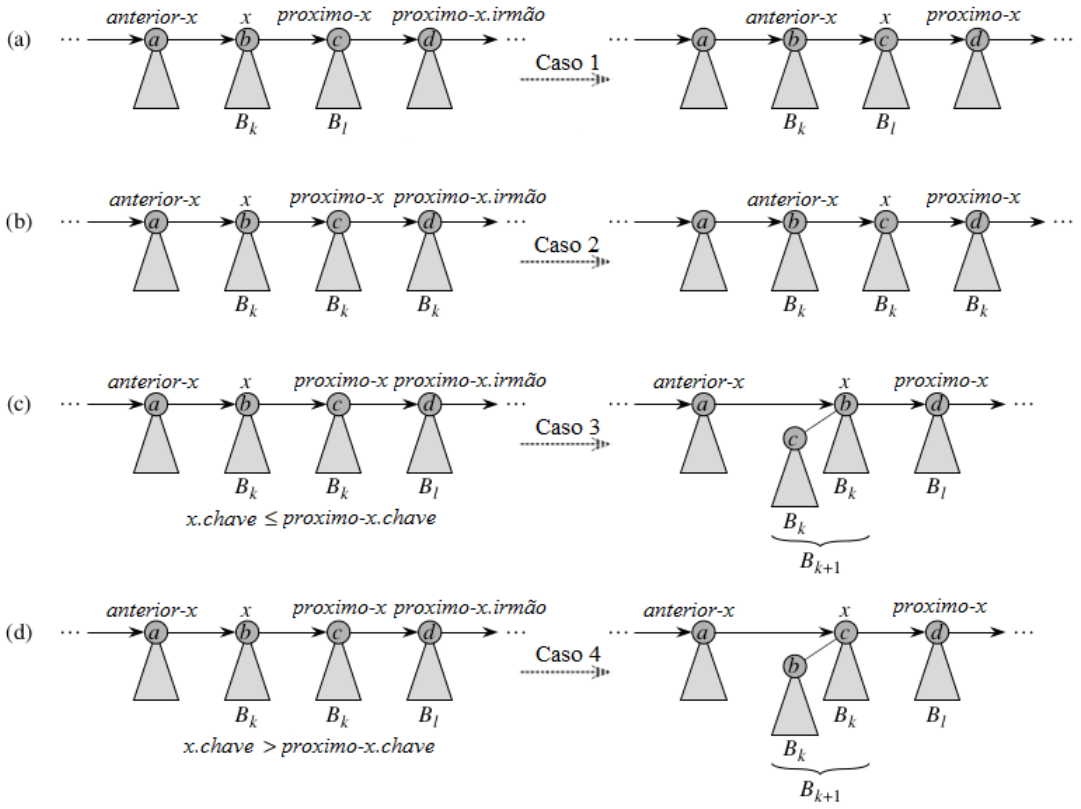

Figura 4.5: Ilustração para os quatro casos possíveis no algoritmo de união. Em **(a)**, as árvores com raízes em `x` e `proximo-x` possuem graus diferentes, então os ponteiros `x`, `anterior-x` e `proximo-x` são simplesmente atualizados (caso 1). Em **(b)**, como as árvores com raízes apontadas por `x`, `proximo-x` e `proximo-x.irmão` possuem o mesmo grau, então simplesmente atualizamos a posição dos ponteiros (caso 2). Em **(c)**, temos que as árvores apontadas por `x` e `proximo-x` possuem o mesmo grau, enquanto as árvores apontadas por `proximo-x` e `proximo-x.irmão` possuem graus diferentes. Logo, realizamos a união entre as árvores apontadas por `x` e `proximo-x`, e como a chave presente em `x` é menor ou igual a chave presente em `proximo-x`, tornamos `proximo-x` o filho mais à esquerda de `x` (caso 3). Em **(d)**, a mesma situação descrita no caso (c), mas nesse caso a chave presente em `x` é maior do que a chave presente em `proximo-x`. Neste caso, `x` se torna o filho mais à esquerda de `proximo-x` (caso 4). Fonte: CORMEN, Thomas et al. Introduction to Algorithms, 2nd edition, The MIT Press. (adaptado)

4.4.4 Inserindo uma chave

O algoritmo `BINOMIAL-HEAP-INSERT` recebe como argumentos um heap binomial H, um nó x com seus campos `x.chave` e `x.valor` previamente preenchidos e insere x em H.

Algorithm 27

```
1: procedure BINOMIAL-HEAP-INSERT(H, x)
2:     H' ← MAKE-BINOMIAL-HEAP()
3:     x.pai ← NULL
4:     x.filho ← NULL
5:     x.irmão ← NULL
6:     x.grau ← 0
7:     H'.inicio ← x
8:     H ← BINOMIAL-HEAP-UNION(H, H')
```

O algoritmo cria um novo heap binomial H' (linha 2), preenche os demais campos de x adequadamente (linhas 3-6), e define x como início da lista de raízes de H' (linha 7). Em seguida, o algoritmo realiza uma operação de união entre o heap binomial H e o novo heap binomial H' (linha 8).

Se o heap binomial H possui n elementos, então o tempo consumido para realizar a operação de

união $H \cup H'$ é $\mathcal{O}(\lg n)$. Logo, a complexidade de tempo do algoritmo `BINOMIAL-HEAP-INSERT` também é $\mathcal{O}(\lg n)$, porque as demais operações são realizadas em tempo constante.

4.4.5 Extraindo o nó com a chave mínima

O algoritmo `BINOMIAL-HEAP-EXTRACT-MIN` recebe como argumento um heap binomial H, e retorna o nó x que contém a chave mínima, removendo x de H.

Algorithm 28

```
1:  procedure BINOMIAL-HEAP-EXTRACT-MIN(H)
2:      if H.inicio = NULL then return NULL
3:      min ← p ← H.inicio
4:      min-anterior ← NULL
5:      while p.irmão ≠ NULL do
6:          if p.irmão.chave < min.chave then
7:              min ← p.irmão
8:              min-anterior ← p
9:          p ← p.irmão
10:     if min-anterior ≠ NULL then
11:         min-anterior.irmão ← min-anterior.irmão.irmão
12:     else H.inicio ← H.inicio.irmão
13:     H' ← MAKE-BINOMIAL-HEAP()
14:     anterior ← NULL
15:     atual ← min.filho
16:     while atual ≠ NULL do
17:         proximo ← atual.irmão
18:         atual.pai ← NULL
19:         atual.irmão ← anterior
20:         anterior ← atual
21:         atual ← proximo
22:     H'.inicio ← atual
23:     H ← BINOMIAL-HEAP-UNION(H, H')
24:     return min
```

Se o heap binomial H estiver vazio (linha 2), então o algoritmo simplesmente retorna `NULL`. Caso contrário, é realizada uma busca na lista de raízes de H, pois o mínimo certamente está ali (linhas 3-9). O nó que contém a chave mínima é armazenado na variável `min`, e seu predecessor é armazenado na variável `min-anterior`. Se o nó que contém a chave mínima for a cabeça da lista de raízes de H (linha 12), então a cabeça da lista de raízes é atualizada para o irmão de `min` (linha 13). De outra forma, o predecessor de `min` é unido ao sucessor de `min`, removendo `min` da lista de raízes de H (linhas 10-11). Em seguida, é criado um novo heap binomial H' (linha 13). As linhas 14-21 invertem a lista ligada contendo os filhos de `min`, pois as árvores binomiais ali presentes estão em ordem decrescente, e define o resultado desta inversão como sendo o início da lista de raízes de H'. Por fim, é realizada uma operação de união entre os heaps binomiais H e H' (linha 23), e o nó contendo a chave mínima é retornado (linha 24).

Como a lista de raízes do heap binomial H possui $\mathcal{O}(\lg n)$ raízes, o loop *while* da linha 5 é executado $\mathcal{O}(\lg n)$ vezes. Além disso, sabendo-se que o heap binomial H possui n nós, temos que o maior grau de uma árvore binomial contida em H é $\lfloor \lg n \rfloor$, e portanto o número de filhos da raiz de tal árvore binomial também é $\lfloor \lg n \rfloor$. Desse modo, o loop *while* na linha 16 é executado $\mathcal{O}(\lg n)$ vezes, e portanto o consumo de tempo total do algoritmo `BINOMIAL-HEAP-EXTRACT-MIN` é $\mathcal{O}(\lg n)$.

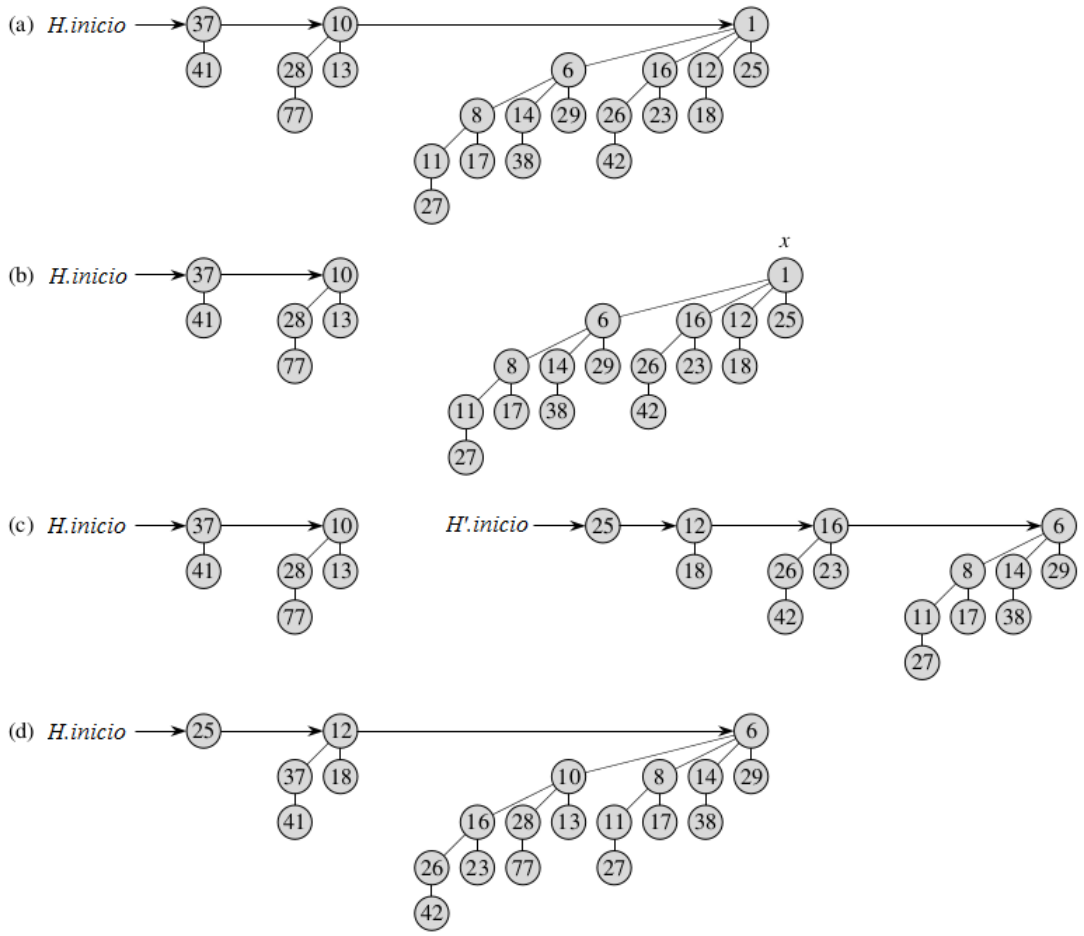

Figura 4.6: Ilustração da remoção do nó mínimo (1) de um heap binomial. Em **(a)**, o heap binomial H na situação inicial. Em **(b)**, o nó x que contém a chave mínima é isolado do heap binomial H. Em **(c)**, os filhos de x constituem um novo heap binomial H'. Em **(d)**, o novo heap H é o resultado da união entre os heaps H antigo e H'. Fonte: CORMEN, Thomas et al. Introduction to Algorithms, 2nd edition, The MIT Press. (adaptado)

4.4.6 Decrementando uma chave

O algoritmo `BINOMIAL-HEAP-DECREASE-KEY` recebe como argumentos um heap binomial H, um nó $x \in H$ e diminui a chave associada a x para k. Caso a chave k seja maior que a chave atualmente associada a x, é lançado um erro e o algoritmo termina.

Algorithm 29

```
1:  procedure BINOMIAL-HEAP-DECREASE-KEY(H, x, k)
2:      if k > x.chave then error: nova chave é maior que a chave atual
3:      x.chave ← k
4:      y ← x
5:      z ← y.pai
6:      while z ≠ NULL e y.chave < z.chave do
7:          y.chave ↔ z.chave
8:          y.valor ↔ z.valor
9:          y ← z
10:         z ← y.pai
```

A linha 3 realiza a atualização da chave em x para k. As linhas 4-10 cuidam da manutenção da propriedade do heap mínimo após a alteração da chave, ou seja, enquanto a chave associada ao nó x for menor que a chave de seu pai, troca-se ambos de posição. O procedimento se repete até que a chave associada a x seja maior que a chave associada ao pai de x.

O número máximo de iterações do loop *while* é limitado por $\lfloor \lg n \rfloor$, porque a maior árvore de um heap binomial que contém n nós possui grau = altura $= \lfloor \lg n \rfloor$. Tendo isso em vista, temos que o consumo de tempo do algoritmo `BINOMIAL-HEAP-DECREASE-KEY` é $\mathcal{O}(\lg n)$.

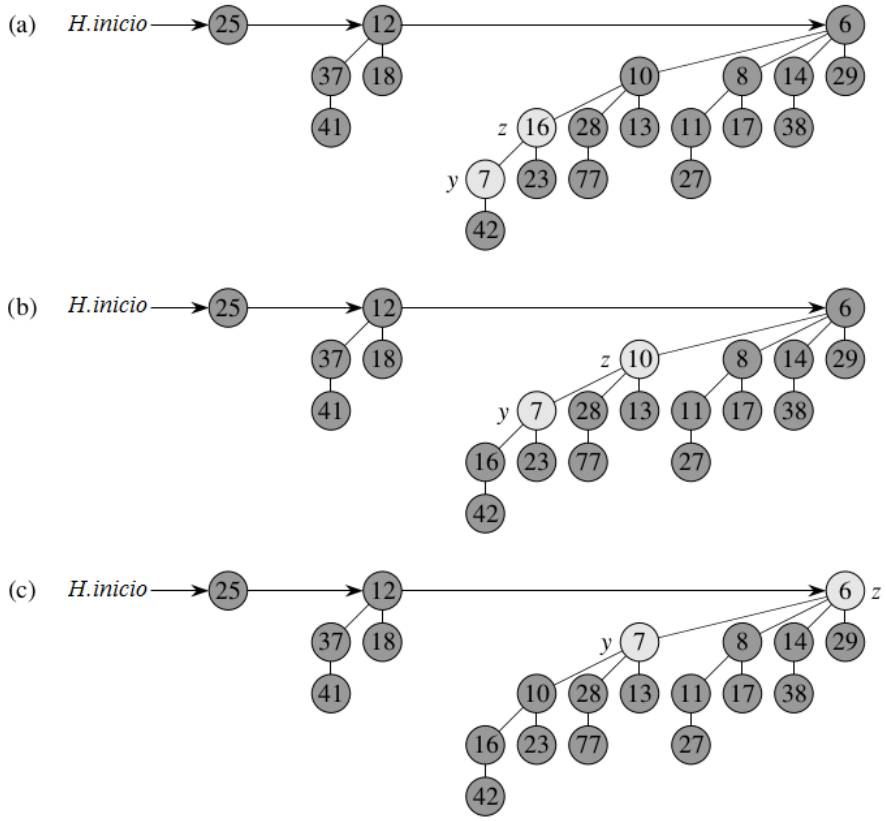

Figura 4.7: Operação `DECREASE-KEY` realizada no heap binomial H. Em **(a)**, a chave 7, associada ao nó y, foi decrementada. Em **(b)**, o nó que contém a chave 7 trocou de lugar com o nó que contém a chave 16. Em **(c)**, o nó que contém a chave 7 trocou de lugar com o nó que contém a chave 10. Neste ponto, não existirão mais trocas, pois a chave 7 é maior que a chave 6. Fonte: CORMEN, Thomas et al. Introduction to Algorithms, 2nd edition, The MIT Press. (adaptado)

4.4.7 Eliminando uma chave

O algoritmo `BINOMIAL-HEAP-DELETE` recebe como argumentos um heap binomial H e um nó x a ser removido de H.

Algorithm 30

1: **procedure** BINOMIAL-HEAP-DELETE(H, x)
2: BINOMIAL-HEAP-DECREASE-KEY(H, x, $-\infty$)
3: BINOMIAL-HEAP-EXTRACT-MIN(H)

Primeiramente, é realizada uma operação do tipo `DECREASE-KEY` em x, para torná-lo o mínimo de H. Em seguida, o nó mínimo é extraído de H, usando-se a operação `EXTRACT-MIN`. Sabemos que este nó é x. Como ambas as operações são $\mathcal{O}(\lg n)$, o consumo de tempo de `BINOMIAL-HEAP-DELETE` é $\mathcal{O}(\lg n)$.

4.5 Usos práticos

Os heaps binomiais são utilizados na prática em algoritmos para escalonamento de processos *round-robin*, bem como em algoritmos para filas de prioridades. Também são utilizados na gerência de redes de alta velocidade.

CAPÍTULO 5

HEAPS DE FIBONACCI

5.1 Introdução

Desenvolvido pelos pesquisadores Michael Fredman e Robert Tarjan, no ano de 1984, os heaps de Fibonacci foram concebidos para terem um bom tempo de execução amortizado para as operações usuais em heaps, com as operações `INSERT`, `FIND-MIN`, `DECREASE-KEY` e `MERGE` possuindo tempos de execução amortizados $\Theta(1)$ (ao invés do tempo $\Theta(\lg n)$, obtido ao usar heaps binomiais), e as operações `EXTRACT-MIN` e `DELETE-MIN` possuindo tempos de execução amortizados $\mathcal{O}(\lg n)$, para um heap de Fibonacci contendo n nós.

Heaps de Fibonacci são muito semelhantes aos heaps binomiais. A principal diferença entre eles é que os heaps de Fibonacci são uma estrutura de dados mais flexível, o que permite adiar o máximo possível a execução de algumas operações, executando-as somente quando forem indispensáveis. Isso viabiliza o tempo de execução $\Theta(1)$ para algumas das operações em heaps de Fibonacci.

Esse tempo de execução otimizado é útil, teoricamente, para melhorar o tempo de execução assintótico de algoritmos importantes, como o algoritmo de Dijkstra. Na prática, porém, é comum o uso de heaps k-ários (heaps em que cada nó possui no máximo k filhos), porque, apesar do tempo de execução de algumas operações em heaps de Fibonacci ser $\Theta(1)$, essa constante oculta é alta, o que impede o uso deste tipo de estrutura de dados em problemas práticos.

5.2 Heaps de Fibonacci

5.2.1 Definição

Um heap de Fibonacci H é um conjunto de árvores, cada uma delas ordenada como um heap mínimo (caso o heap de Fibonacci seja um heap mínimo), com as raízes dessas árvores dispostas em uma lista duplamente ligada, denominada ***a lista de raízes*** de H.

Os campos presentes em um heap de Fibonacci são os seguintes:

- `H.min`: ponteiro para o nó contendo a chave mínima do heap de Fibonacci (mínimo da lista de raízes)
- `H.n`: número de nós presentes no heap de Fibonacci

Cada nó x presente no heap de Fibonacci conterá os seguintes campos:

- `x.chave`: chave associada ao nó x
- `x.valor`: valor associado ao nó x e a chave `x.chave`
- `x.pai`: ponteiro para o nó pai de x; valerá `NULL` caso x seja a raiz de alguma árvore presente no heap de Fibonacci H.

- `x.filho`: ponteiro para qualquer um dos nós filhos de x (presente na lista duplamente ligada, denominada ***a lista de filhos*** de x).
- `x.esquerdo`: ponteiro para o irmão esquerdo de x; caso x não possua irmãos, $x =$ `x.esquerdo`.
- `x.direito`: ponteiro para o irmão direito de x; caso x não possua irmãos, $x =$ `x.direito`.
- `x.grau`: número de nós filhos de x.

Dizemos que um nó x está marcado quando o nó x perder um de seus filhos, após x se tornar um filho de outro nó. O campo que indicará se o nó x está marcado é o campo `x.marca`. Assim sendo, o nó x está marcado se, e somente se, `x.marca` = `TRUE`. Nós recém criados, ou que se tornaram filhos de outros nós, terão este campo imediatamente valendo `FALSE`. A marcação de nós é importante, como veremos em breve, para garantir o limite de tempo assintótico exposto anteriormente para a operação `DECREASE-KEY`.

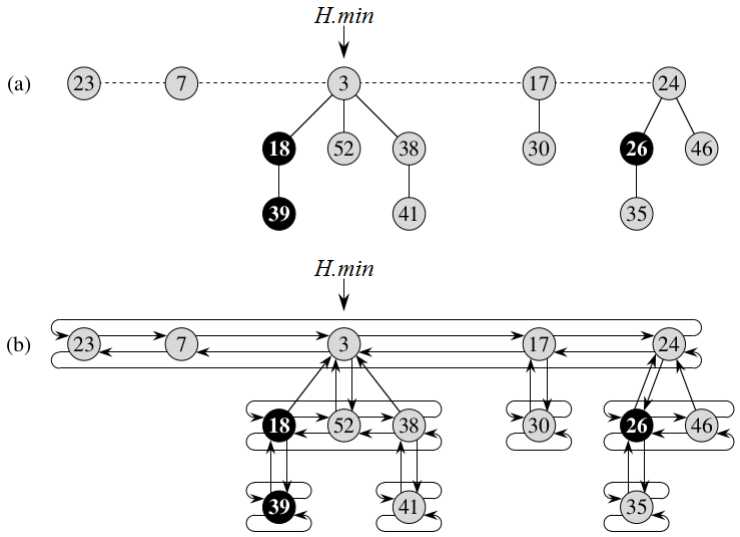

Figura 5.1: **(a)** Um heap de Fibonacci H, cujo campo `H.min` aponta para o nó cuja chave é mínima. **(b)** O mesmo heap de Fibonacci, explicitando-se os ponteiros e as estruturas de listas duplamente ligadas. Fonte: CORMEN, Thomas et al. Introduction to Algorithms, 2nd edition, The MIT Press. (adaptado)

5.2.2 Função potencial

Seja $t(H)$ o número de árvores na lista de raízes do heap de Fibonacci H, e $m(H)$ o número de nós marcados no heap de Fibonacci H. Definimos a função potencial $\Phi(H)$ como segue:

$$\Phi(H) = t(H) + 2m(H)$$

Definimos também o potencial de um conjunto de heaps de Fibonacci como sendo a soma dos potenciais de seus heaps de Fibonacci constituintes.

A função $\Phi(H)$ é sempre não-negativa, uma vez que o potencial de um heap de Fibonacci vazio é 0, enquanto o potencial de qualquer heap de Fibonacci não vazio é maior que 0, pois $t(H) > 0$. Portanto, podemos obter um custo amortizado para as operações, a partir do custo real, usando a função potencial Φ, ou seja:

$$\text{custo amortizado} = \text{custo real} + \Delta\Phi$$

5.2.3 Grau máximo

Durante a análise amortizada das diversas operações sobre heaps de Fibonacci, iremos considerar que existe um limite superior $D(n)$ sobre o grau máximo de qualquer nó em um heap de Fibonacci contendo n nós. Em uma seção futura, mostraremos que $D(n) = \mathcal{O}(\lg n)$.

5.3 Operações usuais

5.3.1 Criando um novo heap de Fibonacci

O procedimento `MAKE-FIB-HEAP` abaixo, ao ser invocado, retorna um heap de Fibonacci vazio. O procedimento pressupõe que uma chamada a `ALLOCATE-MEMORY` cuida da parte de alocação de memória. Em seguida, os campos `H.min` e `H.n` são preenchidos adequadamente, e o novo heap de Fibonacci é retornado. O consumo de tempo do procedimento `MAKE-FIB-HEAP` é $\Theta(1)$.

Algorithm 31

```
1: procedure MAKE-FIB-HEAP()
2:     H ← ALLOCATE-MEMORY()
3:     H.min ← NULL
4:     H.n ← 0
5:     return H
```

5.3.2 Inserindo um novo nó

O procedimento `FIB-HEAP-INSERT` abaixo recebe como argumentos um heap de Fibonacci H, um nó x e o insere em H. O procedimento supõe que os campos `x.chave` e `x.valor` foram inicializados previamente, e que `x.chave` não está presente no heap de Fibonacci H.

Algorithm 32

```
1: procedure FIB-HEAP-INSERT(H, x)
2:     x.grau ← 0
3:     x.pai ← NULL
4:     x.filho ← NULL
5:     x.marca ← FALSE
6:     if H.min = NULL then
7:         x.esquerdo ← x.direito ← x
8:         H.min ← x
9:     else
10:        H.min.esquerdo.direito ← x
11:        x.esquerdo ← H.min.esquerdo
12:        H.min.esquerdo ← x
13:        x.direito ← H.min
14:    if x.chave < H.min.chave then
15:        H.min ← x
16:    H.n ← H.n + 1
```

As linhas 2-5 inicializam os demais campos do nó x adequadamente. Em seguida, a linha 6 verifica se o heap de Fibonacci está vazio; caso seja esse o caso, define-se `H.min` como sendo x (linhas 7-8). Caso contrário (linha 9), as linhas 10-13 inserem x na lista de raízes de H, atualizando os ponteiros conforme necessário. Em seguida, é verificado na linha 14 se a chave associada ao nó x é menor que a chave associada ao nó mínimo atual de H. Se for este o caso, então o novo nó mínimo de H passará a ser o nó x (linha 15). Por fim, na linha 16, o campo `H.n` é incrementado, para refletir as alterações na estrutura.

O custo real do procedimento `FIB-HEAP-INSERT` é $\Theta(1)$. O potencial antes do nó x ser inserido é $t(H) + 2m(H)$, e o potencial após o nó x ser inserido é $(t(H) + 1) + 2m(H)$. Portanto, a variação de potencial $\Delta\Phi$ vale 1, e o custo amortizado da operação `FIB-HEAP-INSERT` é $\Theta(1) + 1 = \Theta(1)$.

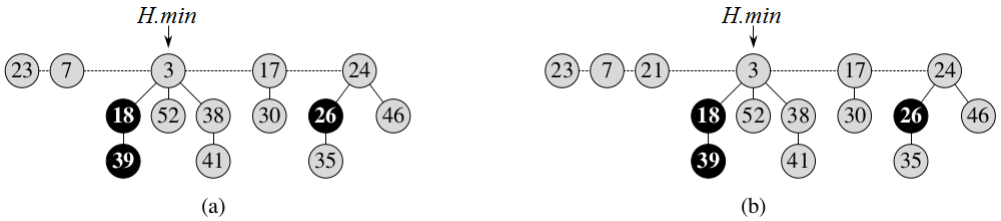

Figura 5.2: **(a)** Um heap de Fibonacci, antes da inserção da chave 21. **(b)** O mesmo heap de Fibonacci, após a inserção da chave 21. Fonte: CORMEN, Thomas et al. Introduction to Algorithms, 2nd edition, The MIT Press. (adaptado)

5.3.3 Encontrando o nó mínimo

O procedimento `FIB-HEAP-MIN` abaixo simplesmente retorna o nó `H.min`. Possui custo real e amortizado $\Theta(1)$, pois não há mudança de potencial.

Algorithm 33

```
1: procedure FIB-HEAP-MIN(H)
2:     return H.min
```

5.3.4 Extraindo o nó mínimo

O procedimento `FIB-HEAP-EXTRACT-MIN` abaixo recebe como argumento um heap de Fibonacci H, remove o nó contendo a chave mínima de tal heap e o retorna.

Algorithm 34

```
1: procedure FIB-HEAP-EXTRACT-MIN(H)
2:     z ← H.min
3:     if z ≠ NULL then
4:         if z.grau ≠ 0 then
5:             for cada filho x de z do
6:                 x.pai ← NULL
7:             esquerdo-antigo ← H.min.esquerdo
8:             z.filho.esquerdo.direito ← H.min
9:             H.min.esquerdo ← z.filho.esquerdo
10:            z.filho.esquerdo ← esquerdo-antigo
11:            esquerdo-antigo.direito ← z.filho
12:        if z = z.direito then H.min ← NULL
13:        else
14:            H.min ← z.direito
15:            z.direito.esquerdo ← z.esquerdo
16:            z.esquerdo.direito ← z.direito
17:            CONSOLIDATE(H)
18:        H.n ← H.n - 1
19:    return z
```

O algoritmo trata vários casos, a saber:

- Se o heap de Fibonacci H não estiver vazio (linha 3), e o nó mínimo possuir filhos (linha 4), então inserimos todos os filhos do nó mínimo z na lista de raízes de H adequadamente, acertando os ponteiros conforme necessário (linhas 5-11). Em seguida, as linhas 13-17 removem z da lista de raízes de H, defininindo o mínimo do heap de Fibonacci H como sendo `z.direito`, temporariamente. A função `CONSOLIDATE` (que iremos analisar em breve) une, sucessivamente aos pares, as árvores pertencentes a H de mesmo grau, para que, ao final do procedimento, a lista de raízes de H possua no máximo uma árvore de cada grau. Isto é necessário para encontrarmos o novo nó mínimo de H.

- Se o heap de Fibonacci H não estiver vazio (linha 3), e o nó mínimo z não possuir filhos, então simplesmente removemos z da lista de raízes adequadamente; as linhas 12-17 cuidam deste caso: se z for o único nó do heap, então a linha 12 acerta o ponteiro `H.min`; e se houverem outras raízes na lista de raízes, as linhas 13-17 removem o nó z da lista de raízes, invocando a função `CONSOLIDATE` em seguida.
- Se o heap de Fibonacci estiver vazio, então o algoritmo simplesmente retorna $z =$ `NULL`.

Por fim, o número de nós presentes no heap de Fibonacci H é decrementado, e o nó z é retornado (linhas 18-19).

O procedimento `CONSOLIDATE` recebe como argumento um heap de Fibonacci H, e une, aos pares, as raízes das árvores de mesmo grau, até que exista, no máximo, uma raiz de cada grau. Esta operação é necessária para determinarmos qual será o novo nó mínimo, após a remoção do mínimo anterior, pelo procedimento `FIB-HEAP-EXTRACT-MIN`.

Algorithm 35

```
1:  procedure CONSOLIDATE(H)
2:      for i ← 0 to D(H.n) do A[i] ← NULL
3:      for cada nó w da lista de raízes de H do
4:          x ← w
5:          d ← x.grau
6:          while A[d] ≠ NULL do
7:              y ← A[d]
8:              if x.chave > y.chave then trocar x ↔ y
9:              FIB-HEAP-LINK(H, y, x)
10:             A[d] ← NULL
11:             d ← d + 1
12:         A[d] ← x
13:     H.min ← NULL
14:     for i ← 0 to D(H.n) do
15:         if A[i] ≠ NULL then
16:             adicionar A[i] à lista de raízes de H
17:             if H.min = NULL or A[i].chave < H.min.chave then
18:                 H.min ← A[i]
```

Primeiramente, a linha 2 inicializa um vetor A contendo $D(n)$ posições. Este vetor, após a linha 12, conterá uma árvore de grau i na posição `A[`i`]`, caso a mesma exista, pra todo $0 \leqslant i \leqslant D(n)$. Se, ao final do processo, não existir uma árvore de grau i, então `A[`i`]` = `NULL`. Em seguida, para cada raiz w pertencente à lista de raízes de H (linha 3), o loop *while* da linha 6 procura por outra raiz de mesmo grau `d` = `w.grau`. Caso tal raiz exista (linha 6), então a raiz y (raiz que possui a maior chave, dentre w e `A[`d`]`) se torna um filho de x (raiz que possui a menor chave, dentre w e `A[`d`]`), usando-se o procedimento `FIB-HEAP-LINK` (que será explicado mais adiante). O loop *while* se repete, após o incremento do valor de d, enquanto `A[`d`]` $\neq$ `NULL`. Quando `A[`d`]` = `NULL`, o nó x é gravado na posição `A[`d`]`, e o loop *for* prossegue para a próxima iteração. Quando todas as raízes w de H forem processadas, o vetor `A` conterá uma raiz de grau i na posição `A[`i`]`, caso exista, ou `NULL` caso contrário. Finalmente, a linha 13 limpa a lista de raízes de H, e o loop *for* da linha 14 percorre o vetor `A`, inserindo as raízes ali presentes na lista de raízes de H. Ao final desse loop *for*, o ponteiro `H.min` apontará para o novo nó mínimo de H.

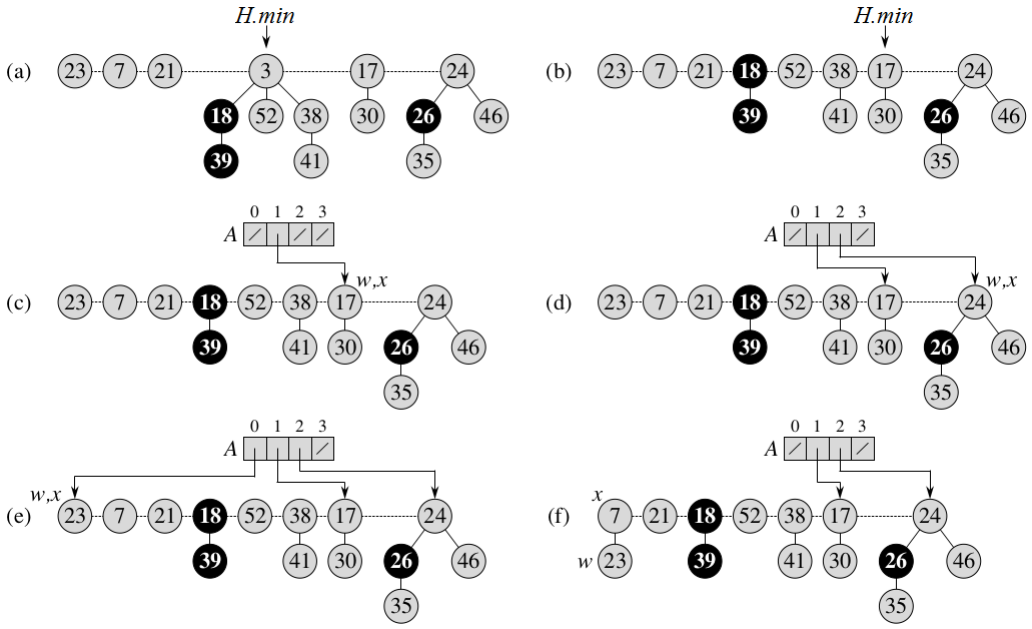

Figura 5.3: Funcionamento do procedimento `CONSOLIDATE`, explicitando o vetor `A`, bem como os ponteiros x e w. Em **(a)**, um heap de Fibonacci H em sua situação inicial. Em **(b)**, o nó mínimo (3) foi removido, e seus filhos foram inseridos na lista de raízes de H. O ponteiro `H.min` foi deslocado para a direita, temporariamente. Em **(c)**, a primeira iteração do loop *for* do algoritmo `CONSOLIDATE`. Em **(d)**, a segunda iteração do mesmo loop. Em **(e)**, a terceira iteração. Em **(f)**, na quarta iteração do loop *for*, na primeira iteração do loop *while*, ocorre a ligação entre os nós 23 e 7. Fonte: CORMEN, Thomas et al. Introduction to Algorithms, 2nd edition, The MIT Press. (adaptado)

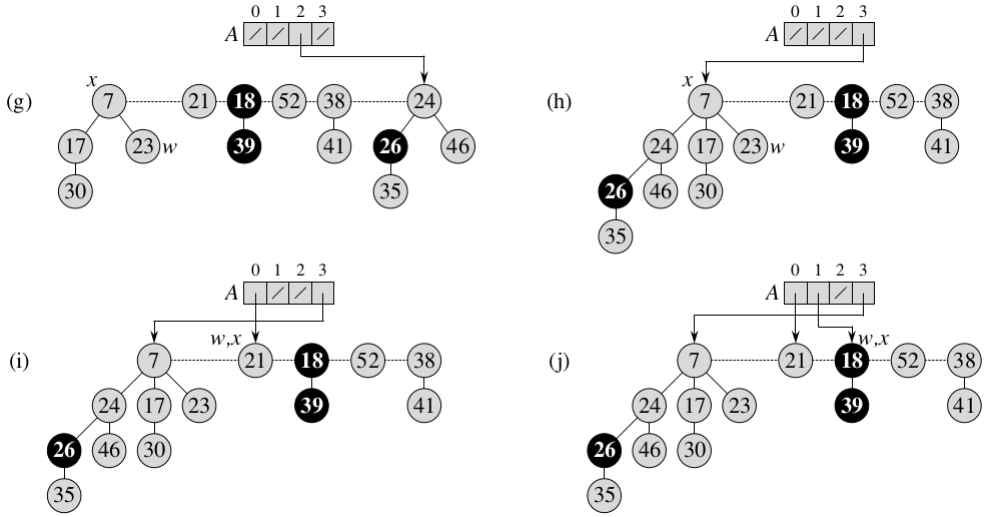

Figura 5.4: Em **(g)**, na quarta iteração do loop *for*, na segunda iteração do loop *while*, o nó com chave 17 foi ligado ao nó com chave 7. Em **(h)**, na quarta iteração do loop *for*, na terceira iteração do loop *while*, o nó com a chave 24 foi ligado ao nó com chave 7. Em **(i)**, a quinta iteração do loop *for*. Em **(j)**, a sexta iteração do loop *for*. Fonte: CORMEN, Thomas et al. Introduction to Algorithms, 2nd edition, The MIT Press.

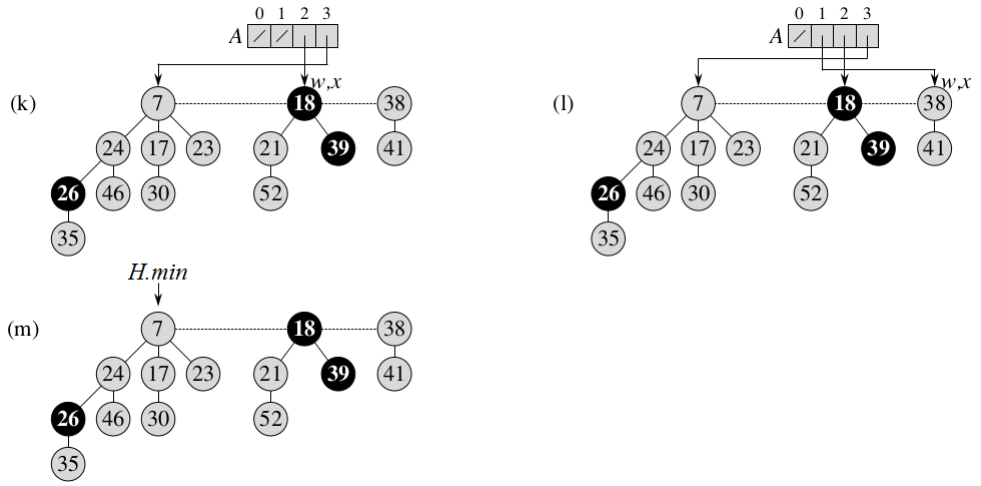

Figura 5.5: Em **(k)**, a sétima iteração do loop *for*. Em **(l)**, a oitava iteração do loop *for*. Em **(m)**, o heap de Fibonacci, após sua reconstrução a partir do vetor A. Fonte: CORMEN, Thomas et al. Introduction to Algorithms, 2nd edition, The MIT Press. (adaptado)

O procedimento FIB−HEAP−LINK recebe como argumentos um heap de Fibonacci H e dois nós x e y, pertencentes à lista de raízes de H, e torna y um filho de x, removendo y da lista de raízes de H.

Algorithm 36

```
1:  procedure FIB-HEAP-LINK(H, y, x)
2:      y.esquerdo.direito ← y.direito
3:      y.direito.esquerdo ← y.esquerdo
4:      if x.filho ≠ NULL then
5:          y.pai ← x
6:          y.direito ← x.filho
7:          y.esquerdo ← x.filho.esquerdo
8:          x.filho.esquerdo ← y
9:          x.filho ← y
10:     else
11:         y.esquerdo ← y.direito ← y
12:         y.pai ← x
13:         x.filho ← y
14:     x.grau ← x.grau + 1
15:     y.marca ← FALSE
```

As linhas 2-3 removem y da lista de raízes de H. Em seguida, as linhas 4-13 tornam y um filho de x. Caso x já possua filhos (linha 4), então as linhas 5-9 inserem y na lista de filhos de x. Caso contrário (linha 10), o procedimento torna y o único filho de x (linhas 11-13). Por fim, o grau de x é incrementado, e a marca de y é definida como FALSE (linhas 14-15).

Agora que entendemos o funcionamento de todos os procedimentos, podemos analisar o custo amortizado de uma chamada a FIB−HEAP−EXTRACT−MIN. O loop *for* da linha 5 de FIB−HEAP−EXTRACT−MIN, e os loops *for* das linhas 2 e 14 de CONSOLIDATE, consomem tempo $\mathcal{O}(D(n))$. Além disso, no momento em que o loop *for* de CONSOLIDATE, na linha 3, é executado, temos no máximo $t(H)+D(n)-1$ nós na lista de raízes de H, pois antes de executar FIB−HEAP−EXTRACT−MIN, tínhamos $t(H)$ nós, e ao remover o antigo nó mínimo, decrementamos em uma unidade tal valor, e acrescentamos no máximo $D(n)$ novos nós na lista de raízes de H. Como, a cada iteração do loop *while* de CONSOLIDATE, ligamos dois nós da lista de raízes de mesmo grau, temos que o tempo consumido pelo loop *for* da linha 3 de CONSOLIDATE é proporcional a $D(n)+t(H)$. Logo, o consumo de tempo real para se extrair o nó mínimo é $\mathcal{O}(D(n)+t(H))$.

Além disso, o potencial antes de uma chamada a FIB−HEAP−EXTRACT−MIN é $t(H)+2m(H)$, e o

potencial depois de uma chamada a `FIB-HEAP-EXTRACT-MIN` é no máximo $(D(n) + 1) + 2m(H)$, porque nenhum nó foi marcado ou desmarcado durante o processo de extração de mínimo, e existirão no máximo $D(n) + 1$ nós na lista de raízes resultante (pois tal lista conterá, no máximo, uma raiz de cada grau, e existe o limite superior $D(n)$ para o grau máximo de um nó). Portanto, o custo amortizado para uma operação `FIB-HEAP-EXTRACT-MIN` é:

$$\begin{aligned} \text{custo amortizado} &= \mathcal{O}(D(n) + t(H)) + ((D(n) + 1) + 2m(H)) - (t(H) + 2m(H)) \\ &= \mathcal{O}(D(n)) + \mathcal{O}(t(H)) - t(H) \\ &= \mathcal{O}(D(n)) \end{aligned}$$

Como veremos em breve, $D(n) = \mathcal{O}(\lg n)$, e portanto o custo amortizado de `FIB-HEAP-EXTRACT-MIN` é $\mathcal{O}(\lg n)$.

5.3.5 Unindo dois heaps de Fibonacci

O algoritmo `FIB-HEAP-UNION` recebe dois heaps de Fibonacci H_1, H_2 e realiza a operação de união entre os heaps. Um novo heap de Fibonacci contendo todas as árvores presentes em $H_1 \cup H_2$ é retornado, e a representação antiga dos heaps H_1 e H_2 é destruída.

Algorithm 37

```
1: procedure FIB-HEAP-UNION(H1, H2)
2:     H ← MAKE-FIB-HEAP()
3:     H.min ← H1.min
4:     concatenar a lista de raízes de H2 com a lista de raízes de H
5:     if H1.min = NULL or (H2.min ≠ NULL and H2.min < H1.min) then
6:         H.min ← H2.min
7:     H.n ← H1.n + H2.n
8:     liberar os objetos H1 e H2
9:     return H
```

A linha 2 cria um novo heap de Fibonacci H, que conterá a união $H_1 \cup H_2$. Em seguida, as linhas 3-6 concatenam a lista de raízes dos heaps de Fibonacci H_1 e H_2, atualizando o ponteiro para o nó mínimo desta nova lista de raízes concatenada. Finalmente, o campo `H.n`, que conta o número de nós presentes no heap H, é inicializado devidamente, as representações dos heaps H_1 e H_2 são destruídas e o novo heap de Fibonacci H é retornado (linhas 7-9).

O consumo de tempo real de `FIB-HEAP-UNION` é $\Theta(1)$, porque a concatenação das listas de raízes pode ser realizada em tempo constante. Além disso, o potencial antes da invocação do procedimento é $(t(H_1) + 2m(H_1)) + (t(H_2) + 2m(H_2))$, e o potencial após a invocação do procedimento é $(t(H_1) + t(H_2)) + 2(m(H_1) + m(H_2))$. Logo, $\Delta\Phi = 0$, e o custo amortizado de `FIB-HEAP-UNION` também é $\Theta(1)$.

5.3.6 Decrementando uma chave

O procedimento `FIB-HEAP-DECREASE-KEY` recebe como argumentos um heap de Fibonacci H, um nó $x \in H$ e um novo valor de chave k para o nó x. A chave de x será atualizada para k somente se `x.chave` $\geqslant k$ no momento da invocação do procedimento. Caso contrário, será lançado um erro.

Algorithm 38

```
1: procedure FIB-HEAP-DECREASE-KEY(H, x, k)
2:     if k > x.chave then error "nova chave é maior que a chave atual"
3:     x.chave ← k
4:     y ← x.pai
5:     if y ≠ NULL and x.chave < y.chave then
6:         CUT(H, x, y)
7:         CASCADING-CUT(H, y)
8:     if x.chave < H.min.chave then
9:         H.min ← x
```

A linha 2 verifica se a chave k é maior que a chave presente no nó x. Se for, então o procedimento lança um erro e termina. Caso contrário, a linha 3 atualiza a chave presente em x para k, e a linha 4 atribui a y o nó pai de x. Se $y \neq$ `NULL`, então é necessário verificar se a alteração na chave de x violou as propriedades de uma árvore ordenada como heap mínimo. Caso a chave de x seja menor que a chave de y, então cortamos o nó x de y, inserindo-o na lista de raízes de H (linhas 5-7).

Um nó qualquer será cortado quando:

- violar a propriedade do heap mínimo, como visto acima, ou
- perder dois filhos, desde que se tornou filho de um outro nó; tal nó estará marcado antes da perda do segundo filho.

A operação de corte é realizada pelo procedimento `CUT`, que será analisado em breve. Após removermos o filho x de y, pode ser que y precise ser cortado também, por ser um nó marcado, ou seja, y está perdendo seu segundo filho x desde que se tornou filho de outro nó. Essa verificação é realizada pelo procedimento recursivo `CASCADING-CUT`, que será analisado em breve. Tal procedimento "sobe" pela árvore, cortando os nós marcados sucessivamente, até encontrar um nó que não está marcado, ou a raiz da árvore que contém y. Por fim, as linhas 8-9 atualizam, conforme necessário, o ponteiro `H.min`. Como x foi o único nó no heap de Fibonacci que sofreu alteração, temos que o novo nó mínimo é o nó mínimo antigo, ou x.

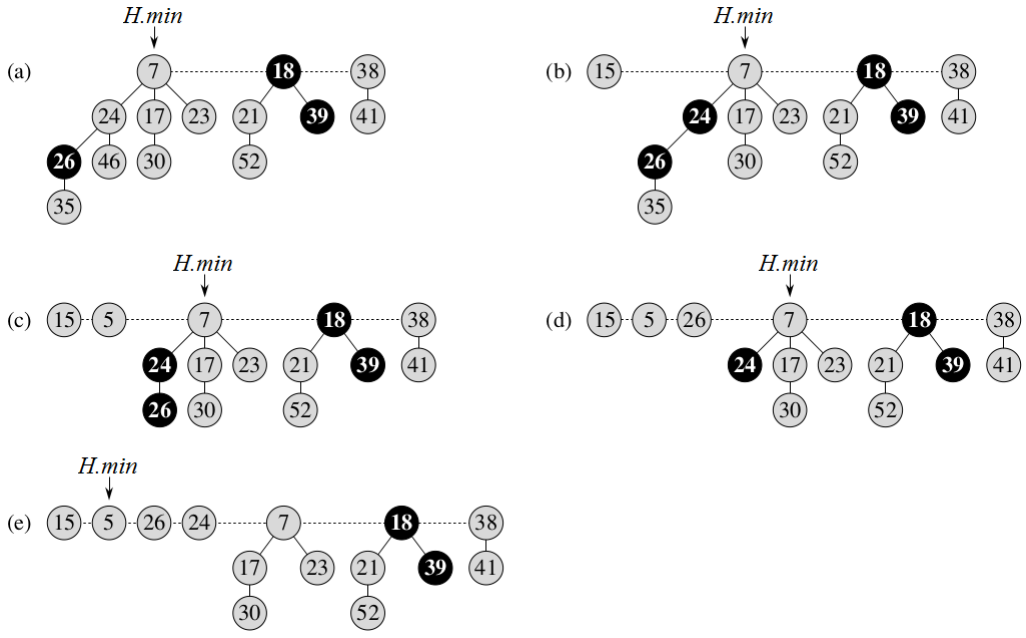

Figura 5.6: Operação `DECREASE-KEY`. Em **(a)**, um heap de Fibonacci H, em sua situação inicial. Em **(b)**, a chave 46 foi decrementada para 15, portanto o nó associado foi cortado. Seu pai, o nó 24, foi marcado. Em **(c)**, a chave 35 foi decrementada para 5, e seu nó associado foi cortado. Seu pai, que já estava marcado, será cortado também. Em **(d)**, corte do nó cuja chave é 26. Como seu pai (24) também está marcado, o corte se propaga até ele. Em **(e)**, corte do nó cuja chave é 24. Cmo seu pai (7) não está marcado, a propagação dos cortes pela árvore chega ao fim. Fonte: CORMEN, Thomas et al. Introduction to Algorithms, 2nd edition, The MIT Press. (adaptado)

O procedimento `CUT` recebe como argumentos um heap de Fibonacci H e dois nós x e y tais que x é filho de y. Remove x da lista de filhos de y e o insere na lista de raízes de H.

Algorithm 39

```
1:  procedure CUT(H, x, y)
2:      if y.filho = x then
3:          if x.direito = x then y.filho ← NULL
4:          else x.esquerdo.direito ← x.direito
5:               x.direito.esquerdo ← x.esquerdo
6:               y.filho ← x.direito
7:      else x.esquerdo.direito ← x.direito
8:           x.direito.esquerdo ← x.esquerdo
9:      x.direito ← H.min
10:     x.esquerdo ← H.min.esquerdo
11:     H.min.esquerdo.direito ← x
12:     H.min.esquerdo ← x
13:     x.pai ← NULL
14:     x.marca ← FALSE
15:     y.grau ← y.grau - 1
```

As linhas 2-8 removem x da lista de filhos de y. Caso x seja o único filho de y (linha 3), então `y.filho` é definido como `NULL`. Caso `y.filho` aponte para o nó x, e x possua irmãos (linha 4), então as linhas 4-6 removem x da lista de filhos de y, ajustando os ponteiros conforme necessário. Caso contrário, simplesmente removemos x da lista de filhos de y (linhas 7-8). Em seguida, inserimos x na lista de raízes de H (linhas 9-13), removemos a marca do nó x, caso exista (linha 14) e decrementamos o grau de y (linha 15).

O procedimento `CASCADING-CUT` recebe um heap de Fibonacci H e um nó y, que acabou de perder um filho. A finalidade deste procedimento é remover todos os nós que estão marcados recursivamente, até encontrar um nó que não esteja marcado ou chegar à raiz da árvore que contém y.

Algorithm 40

```
1:  procedure CASCADING-CUT(H, y)
2:      z ← y.pai
3:      if z ≠ NULL then
4:          if not y.marca then
5:              y.marca ← TRUE
6:          else CUT(H, y, z)
7:               CASCADING-CUT(H, z)
```

A linha 2 define z como o pai de y. Se $z \neq$ `NULL` (linha 3), e o nó y não estiver marcado, então marcamos o nó y e terminamos (linhas 4-5). Caso o nó y já esteja marcado (linha 6), então cortamos tal nó e chamamos recursivamente para o nó z (linhas 6-7). A recursão termina quando encontrarmos um nó que não está marcado, ou quando z for a raiz da árvore que contém y.

Agora, iremos analisar o consumo de tempo amortizado do procedimento `FIB-HEAP-DECREASE-KEY`. O consumo de tempo de `FIB-HEAP-DECREASE-KEY` é $\Theta(1)$, acrescido do consumo de tempo das chamadas recursivas a `CASCADING-CUT`. Supondo que exista c chamadas recursivas a `CASCADING-CUT`, e tendo em vista que o consumo de tempo de `CUT` é $\Theta(1)$, e que cada encarnação da recursão também consome tempo $\Theta(1)$, temos que o consumo de tempo das chamadas recursivas a `CASCADING-CUT` é $\Theta(c)$. Portanto, o consumo de tempo real do procedimento `FIB-HEAP-DECREASE-KEY` é $\Theta(1) + \Theta(c) = \Theta(c)$.

Agora, iremos analisar a variação de potencial durante a operação DECREASE−KEY. O potencial antes da operação é $t(H)+2m(H)$. O número de nós na lista de raízes de H após a operação é $t(H)+c$, pois $c-1$ nós foram adicionados pelas chamadas recursivas a CASCADING−CUT (a última chamada recursiva não adicionou nenhum nó na lista de raízes de H) e além disso o nó x também foi adicionado à lista de raízes. O número de nós marcados ao final da operação é no máximo $m(H)-(c-1)+1$, pois $c-1$ nós foram cortados durante as chamadas recursivas, e a última chamada recursiva pode ter marcado um nó, ou não. Portanto, o custo amortizado é no máximo:

$$\begin{aligned}\text{custo amortizado} &= \Theta(c)+((t(H)+c)+2(m(H)-(c-1)+1))-(t(H)+2m(H)) \\ &= \Theta(c)-c+4 \\ &= \Theta(1)\end{aligned}$$

5.3.7 Eliminando um nó

O procedimento FIB−HEAP−DELETE recebe um heap de Fibonacci H, um nó x e remove x de H.

Algorithm 41

```
1: procedure FIB-HEAP-DELETE(H, x)
2:     FIB-HEAP-DECREASE-KEY(H, x, −∞)
3:     FIB-HEAP-EXTRACT-MIN(H)
```

A linha 2 decrementa a chave de x para um valor $-\infty$, menor que qualquer chave presente em H. Em seguida, o nó x, que agora é o novo nó mínimo de H, é extraído na linha 3. O custo amortizado de FIB−HEAP−DECREASE−KEY, como visto anteriormente, é $\Theta(1)$, enquanto o custo amortizado de FIB−HEAP−EXTRACT−MIN, conforme iremos ver posteriormente, é $\mathcal{O}(\lg n)$. Portanto, o custo amortizado de FIB−HEAP−DELETE é $\Theta(1)+\mathcal{O}(\lg n)=\mathcal{O}(\lg n)$.

5.4 Limitando o grau máximo de um nó

Nas seções anteriores, assumimos que existe um limite superior $D(n)$ para o grau máximo de um nó presente em um heap de Fibonacci H contendo n nós. Antes de demonstrarmos que $D(n)=\mathcal{O}(\lg n)$, iremos enunciar alguns lemas, que serão fundamentais para a demonstração deste resultado.

Seja a sequência de Fibonacci definida recursivamente como segue:

$$F_k=\begin{cases}0, \text{ se } k=0\\ 1, \text{ se } k=1\\ F_{k-1}+F_{k-2}, \text{ caso contrário}\end{cases}$$

Lema 5.4.1. *Para todo k inteiro positivo,*

$$F_{k+2}=1+\sum_{i=0}^{k}F_i$$

Demonstração. Indução em k. Para $k=0$,

$$F_2=1+F_0=1$$

Vamos mostrar que também vale para $k\geqslant 1$. Supondo que vale para $k-1$, temos:

$$F_{k+1}=1+\sum_{i=0}^{k-1}F_i \quad\Longleftrightarrow\quad F_k+F_{k+1}=F_k+1+\sum_{i=0}^{k-1}F_i \quad\Longleftrightarrow\quad F_{k+2}=1+\sum_{i=0}^{k}F_i$$

□

Lema 5.4.2. *Sejam $y_1, y_2, \ldots, y_k$ os nós filhos de x, na ordem em que se tornaram filhos de x. Então o grau de y_1 é $\geqslant 0$ e o grau de y_i, para $2 \leqslant i \leqslant k$, é maior ou igual a $i-2$.*

Demonstração. O grau de $y_1 \geqslant 0$ é trivial. Considere um filho y_i qualquer. No momento em que este nó se tornou filho de x, haviam outros $y-1$ filhos. Logo, o grau de y_i era igual a $i-1$ naquele momento. Acontece que o nó y_i pode ter perdido no máximo um filho, pois se tivesse perdido dois filhos, teria sido cortado. Logo o seu grau será no mínimo $i-2$. □

Considere a função $\texttt{size}(x)$, que associa a cada nó x de grau k o número total de nós presentes na subárvore com raiz em x.

Lema 5.4.3. $\texttt{size}(x) \geqslant F_{k+2}$.

Demonstração. Seja s_k o menor valor de $\texttt{size}(z)$, para todo nó z de grau k. Sabemos que $s_0 = 1$, $s_1 = 2$ e $s_2 = 3$. Também é possível notar que s_k cresce monotonicamente, conforme k cresce. Devemos mostrar agora que $s_k \geqslant F_{k+2}$, pois a desigualdade $\texttt{size}(x) \geqslant s_k$ é imediata, e a prova será por indução forte em k. Para $k = 0$, temos:

$$1 = s_0 \geqslant F_2 = 1$$

Agora, supondo que

$$s_i \geqslant F_{i+2}$$

para $i \in \{1, 2, \ldots, k-1\}$, devemos mostrar que o resultado também vale para k. Temos que s_k pode ser reescrito como

$$s_k = 2 + \sum_{i=2}^{k} s_{\text{grau}(y_i)}$$

porque contamos uma unidade para o próprio nó x, e outra unidade para o filho y_1 de x. Além do mais, utilizando-se o Lema 5.4.2, como s cresce monotonicamente conforme k cresce, temos:

$$\text{grau}(y_i) \geqslant i-2 \quad \Longrightarrow \quad s_{\text{grau}(y_i)} \geqslant s_{i-2}$$

logo,

$$\sum_{i=2}^{k} s_{\text{grau}(y_i)} \geqslant \sum_{i=2}^{k} s_{i-2}$$

usando-se a hipótese de indução, e o Lema 5.4.1, temos que

$$\sum_{i=2}^{k} s_{i-2} \geqslant \sum_{i=2}^{k} F_i \quad \Longrightarrow \quad 2 + \sum_{i=2}^{k} s_{i-2} \;\geqslant\; 2 + \sum_{i=2}^{k} F_i \;=\; 1 + \sum_{i=0}^{k} F_i \;=\; F_{k+2}$$

logo, $s_k \geqslant F_{k+2}$, e isso mostra que $\texttt{size}(x) \geqslant F_{k+2}$. □

Lema 5.4.4. *Para todo inteiro $k \geqslant 0$, temos que $F_{k+2} \geqslant \varphi^k$, onde $\varphi = \left(1+\sqrt{5}\right)/2$ é o número de ouro.*

Demonstração. A prova será por indução forte em k. Para $k = 0$ e $k = 1$, temos que:

$$\begin{array}{rclcrcl} 1 & = & F_2 & \geqslant & \varphi^0 & = & 1 \\ 2 & = & F_3 & \geqslant & \varphi^1 & = & 1.61803398875\ldots \end{array}$$

Supondo que o resultado vale para $k-2$ e $k-1$, para $k \geqslant 2$, temos:

$$F_k \geqslant \varphi^{k-2},\ F_{k+1} \geqslant \varphi^{k-1} \quad \Longrightarrow \quad F_{k+2} = F_k + F_{k+1} \geqslant \varphi^{k-2} + \varphi^{k-1} = (\varphi + 1)\varphi^{k-2}$$

Usando o fato que $\varphi + 1 = \varphi^2$, finalmente obtemos $F_{k+2} \geqslant \varphi^k$. □

Do exposto acima, podemos finalmente enunciar o seguinte teorema.

Teorema 5.4.5. $D(n) = \mathcal{O}(\lg n)$.

Demonstração. Para qualquer nó x de grau k, e considerando-se os Lemas 5.4.3 e 5.4.4,

$$n \geqslant \texttt{size}(x) \geqslant F_{k+2} \geqslant \varphi^k \quad \Longrightarrow \quad k \leqslant \log_\varphi n$$

Logo, $k = \mathcal{O}(\lg n)$, e isso mostra que $D(n) = \mathcal{O}(\lg n)$. □

A sequência de Fibonacci aparece nas análises feitas anteriormente; por este motivo, a estrutura de dados leva o nome heaps de Fibonacci.

5.5 Usos teóricos

Heaps de Fibonacci são utilizados, teoricamente, para melhorar o tempo de execução assintótico dos algoritmos de Prim, Kruskal, Cheriton-Tarjan e Dijkstra.

CAPÍTULO 6

TRIES

6.1 Introdução

Desenvolvida pelos pesquisadores Edward Fredkin, Axel Thue e René de la Briandais, em 1960, as *tries*, formalmente conhecidas como R*-way tries* e informalmente conhecidas como árvores de prefixos, são um tipo de árvore R-ária, usadas para representar um conjunto de strings. Inicialmente, foram desenvolvidas para serem utilizadas no contexto de recuperação de informações (*retrieval*; é desse termo que surgiu o nome *trie*). As *tries* possuem a interessante propriedade de possuir tempo de busca, inserção e remoção no pior caso $\mathcal{O}(|s|)$, onde $|s|$ é o tamanho da string s que estamos trabalhando. Por outro lado, tal estrutura consome mais memória principal, o que pode não ser vantajoso em alguns cenários.

6.2 Definição

Uma R*-way trie* T é uma árvore de busca sobre um alfabeto de R caracteres. Cada nó de T possui grau máximo igual a R, e cada caminho de T codifica um caractere. A sequência de caracteres, codificada a partir da raiz até determinado nó x, definem uma string s. A string s está associada ao nó x, podendo existir um valor v vinculado a ele. Caso exista, dizemos que s é uma chave, e v o valor associado à string s. A string vazia está associada ao nó raiz de T.

Uma *trie* T conterá os seguintes campos:

- `T.raiz`: ponteiro para o nó raiz da *trie*.
- `T.R`: valor numérico que indica o tamanho do alfabeto que estamos trabalhando.

Além disso, qualquer nó $x \in T$ conterá os seguintes campos:

- `x.proximo`: vetor contendo R posições; cada posição corresponde a uma letra do alfabeto, e `x.proximo[`i`]` $\neq$ `NULL` se, e somente se, existe uma string s' associada ao nó `x.proximo[`i`]`.
- `x.val`: valor associado à string s; esta, por sua vez, associada ao nó x; caso não exista valor associado, valerá `NULL`.

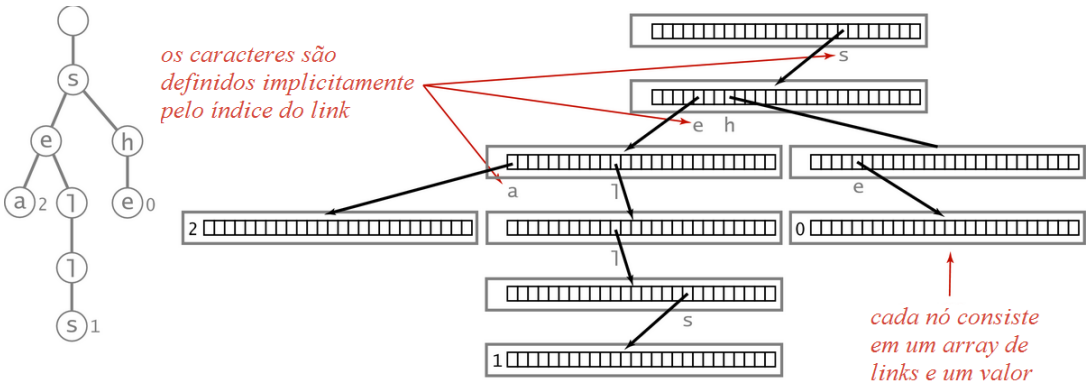

Figura 6.1: Representação de um nó da *trie*. Cada nó consiste em um vetor, com R posições, e um campo `valor`, que armazena o valor associado àquele nó (caso exista um).
Fonte: https://www.ime.usp.br/ pf/estruturas-de-dados/aulas/tries.html (adaptado)

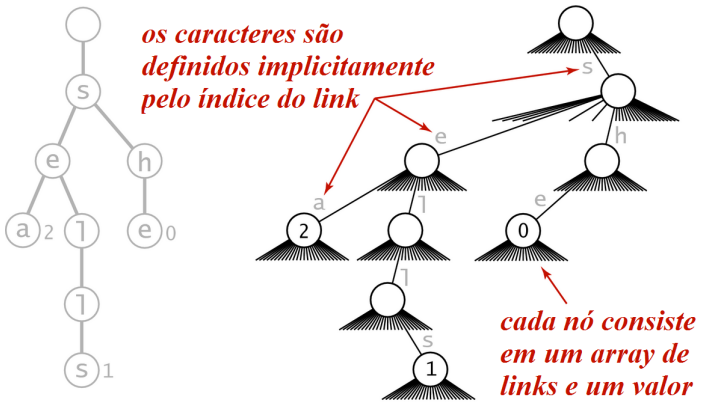

Figura 6.2: Representação compacta da mesma estrutura, explicitando os caminhos entre os nós.
Fonte: https://www.ime.usp.br/ pf/estruturas-de-dados/aulas/tries.html (adaptado)

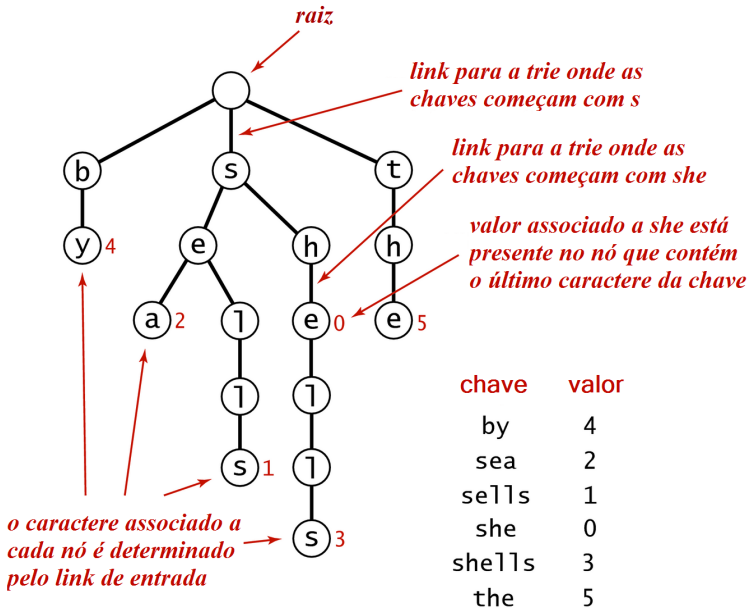

Figura 6.3: Uma *trie*, contendo algumas chaves e valores associados.
Fonte: https://www.ime.usp.br/ pf/estruturas-de-dados/aulas/tries.html (adaptado)

6.3 Operações usuais

6.3.1 Criando uma *trie*

O procedimento `CREATE` abaixo recebe um parâmetro R, o tamanho do alfabeto que iremos trabalhar e, ao ser invocado, retorna uma nova *trie* T, com os seus campos devidamente inicializados.

Algorithm 42

```
1: procedure CREATE(R)
2:     T ← ALLOCATE-MEMORY()
3:     T.raiz ← ALLOCATE-MEMORY()
4:     T.R ← R
5:     return T
```

A linha 2 obtém do sistema subjacente uma quantidade de memória suficiente para armazenar T. Isso é abstraído pela função `ALLOCATE-MEMORY`. Em seguida, a linha 3 atribui a `T.raiz` um novo nó, e a parte que cuida da alocação de memória de tal nó também é abstraída pela função `ALLOCATE-MEMORY`. A linha 4 inicializa o campo `T.R`, e a linha 5 retorna a nova estrutura criada.

O consumo de tempo deste algoritmo é $\Theta(R)$, dado o loop *for* nas linhas 5-6.

Durante o decorrer deste texto, iremos supor que uma chamada a `ALLOCATE-MEMORY` irá entregar tanta memória quanto for necessária, sempre com os bits valendo zero.

6.3.2 Pesquisando uma chave

O procedimento `SEARCH` recebe como argumentos uma *trie* T e uma string k, e retorna o valor associado à chave k, caso exista. Este procedimento usa o procedimento auxiliar `SEARCH-RECURSIVE`, que será explicado em breve.

Algorithm 43

```
1: procedure SEARCH(T, k)
2:     x ← SEARCH-RECURSIVE(T.raiz, k, 1)
3:     if x = NULL then return NULL
4:     return x.val
5:
6: procedure SEARCH-RECURSIVE(x, k, d)
7:     if x = NULL then return NULL
8:     if d = |k| + 1 then return x
9:     c ← k[d]                          ▷ d-ésimo caractere da string k
10:    return SEARCH-RECURSIVE(x.proximo[c], k, d+1)
```

A linha 2 da função `SEARCH` chama o procedimento recursivo `SEARCH-RECURSIVE`, que recebe como argumentos um nó x, uma chave k e um inteiro não negativo d, e retorna o nó associado à chave k, se existir. Caso o valor retornado por `SEARCH-RECURSIVE` seja igual a `NULL` (linha 3), então o procedimento retorna `NULL`; caso contrário, a linha 4 retorna o valor associado à chave k.

O procedimento recursivo `SEARCH-RECURSIVE` recebe como argumentos um nó x, uma chave k e um inteiro d não negativo. A recursão começa com $d = 1$, e a cada chamada recursiva, o procedimento tenta descer um nível na árvore de busca, incrementando o valor de d, a partir do nó x. A recursão acaba quando $d = |k| + 1$ (linha 8), ou quando a recursão for chamada passando-se como argumento $x =$ `NULL` (linha 7). Se a recursão ainda não chegou ao fim, então a linha 9 decide em qual ramo da árvore a recursão deve continuar, com base no caractere correspondente na string k. Em seguida, a recursão continua a partir daquele ramo (linha 10), e caso não exista um nó associado àquela posição no array `x.proximo`, a próxima chamada recursiva irá retornar `NULL`.

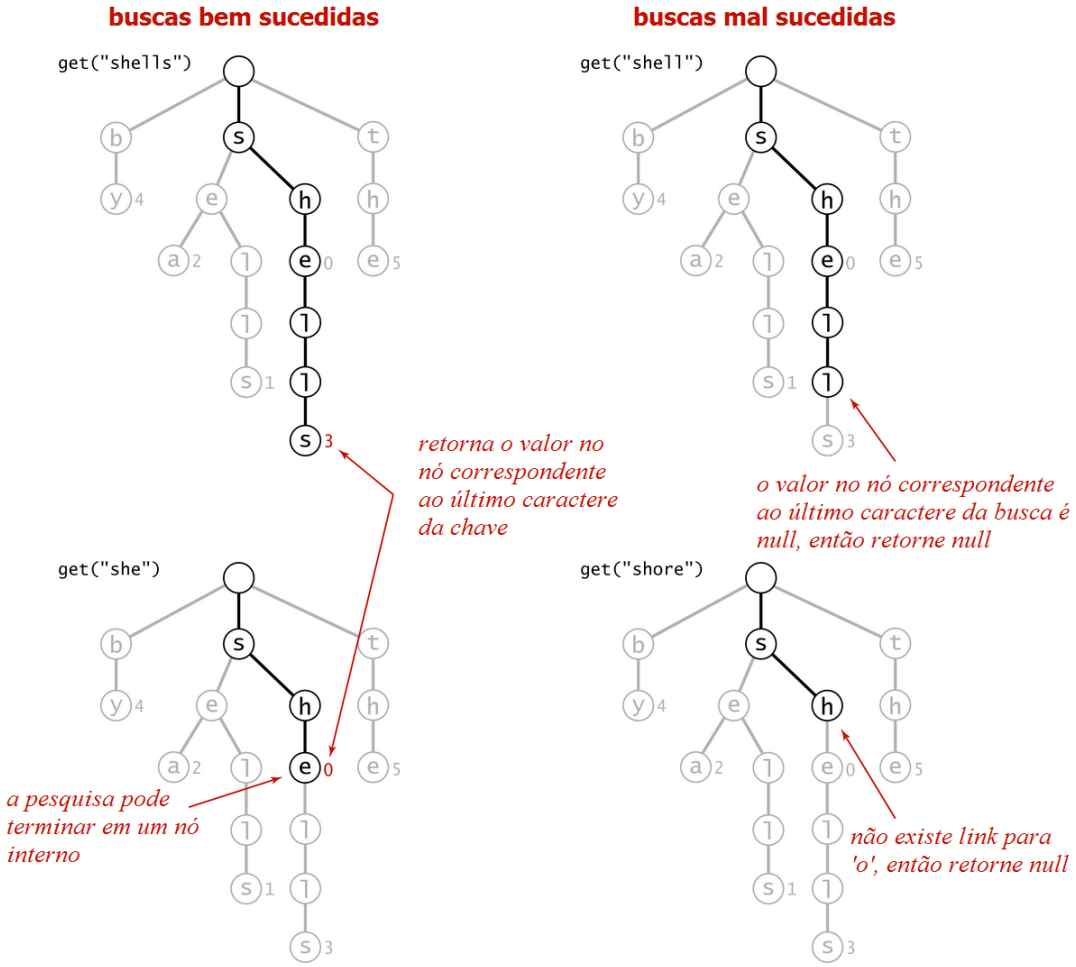

Figura 6.4: Buscas em uma *trie*, explorando-se casos bem sucedidos e mal sucedidos.
Fonte: https://www.ime.usp.br/ pf/estruturas-de-dados/aulas/tries.html (adaptado)

6.3.3 Inserindo uma chave

O procedimento `INSERT` recebe como parâmetros uma *trie* T, uma chave k e um valor v associado à chave k e os inserem na *trie* T. Este procedimento usa o procedimento auxiliar `INSERT-RECURSIVE`, que será explicado em breve.

Algorithm 44

```
1:  procedure INSERT(T, k, v)
2:      T.raiz ← INSERT-RECURSIVE(T.raiz, k, v, 1)
3:
4:  procedure INSERT-RECURSIVE(x, k, v, d)
5:      if x = NULL then x ← ALLOCATE-MEMORY()
6:      if d = |k| + 1 then
7:          x.val ← v
8:          return x
9:      c ← k[d]
10:     x.proximo[c] ← INSERT-RECURSIVE(x.proximo[c], k, v, d+1)
11:     return x
```

A função `INSERT` simplesmente atribui a `T.raiz` o valor retornado por `INSERT-RECURSIVE`. Este último recebe como argumentos um nó x, uma chave k, um valor v associado à chave k e um inteiro não negativo d, inicialmente valendo 1, que será incrementado sucessivamente a cada passo da recursão. A ideia do procedimento é ir criando nós, conforme necessário e recursivamente, até que $d = |k| + 1$.

Neste momento, a recursão associa ao nó x o valor v, e a recursão termina.

A linha 5 cuida do caso em que um nó no caminho que leva até k não existe. O procedimento cria o nó, e as linhas 9-11 continuam a recursão no ramo apropriado, a depender do caractere $k[d]$. Caso $d = |k| + 1$, então associamos v ao nó x (linhas 6-8) e o retornamos na linha 11.

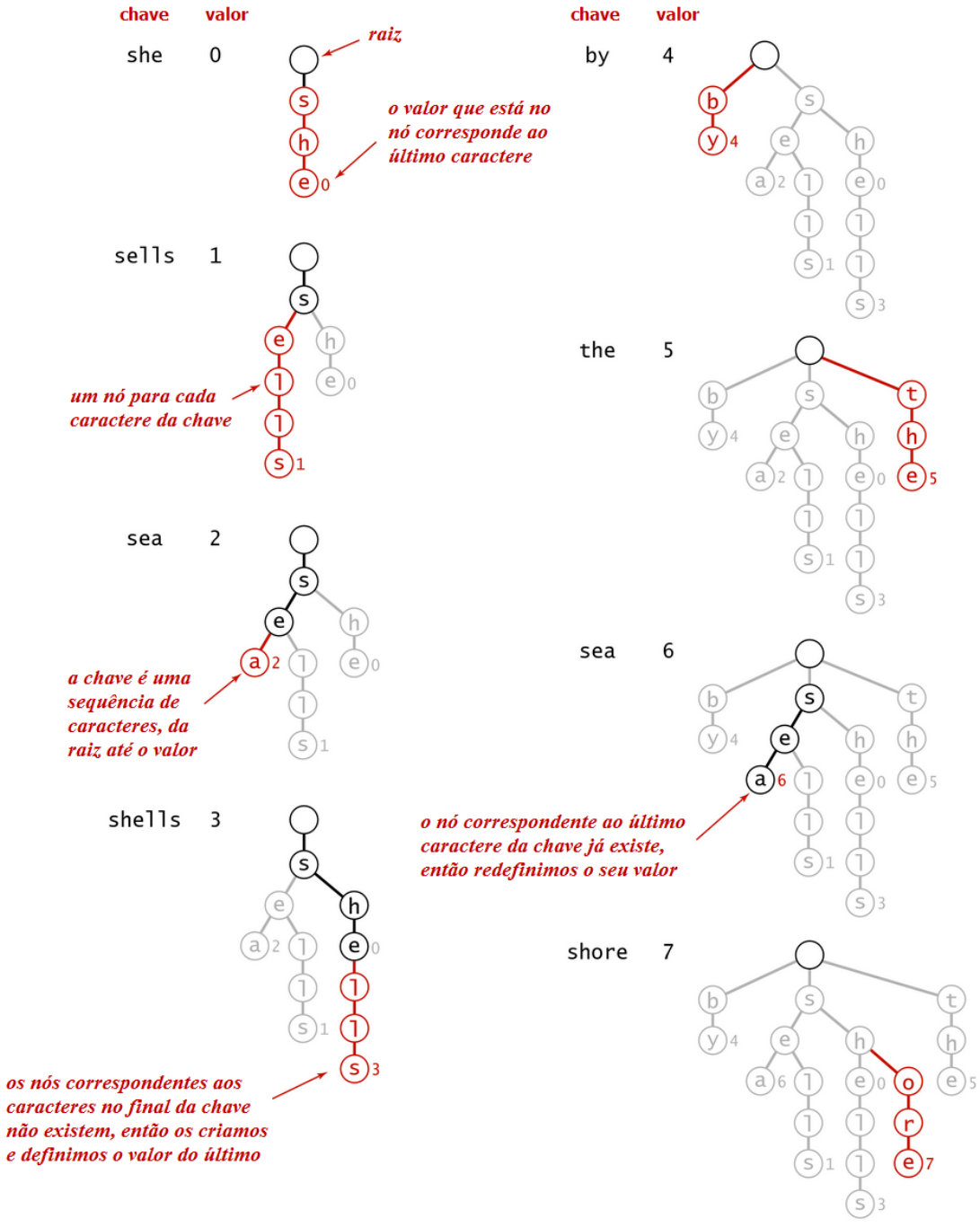

Figura 6.5: Inserção de chaves numa *trie*, em diferentes cenários.
Fonte: https://www.ime.usp.br/ pf/estruturas-de-dados/aulas/tries.html (adaptado)

6.3.4 Removendo uma chave

O procedimento `DELETE` recebe como argumentos uma *trie* T, uma chave k e remove o valor associado à chave k de T, caso exista. Este procedimento usa o procedimento auxiliar `DELETE-RECURSIVE`, que será explicado em breve.

Algorithm 45

```
1:  procedure DELETE(T, k)
2:      T.raiz ← DELETE-RECURSIVE(T, T.raiz, k, 1)
3:
4:  procedure DELETE-RECURSIVE(T, x, k, d)
5:      if x = NULL then return NULL
6:      if d = |k| + 1 then x.val ← NULL
7:      else c ← k[d]
8:          x.proximo[c] ← DELETE-RECURSIVE(x.proximo[c], k, d+1)
9:      if x.val ≠ NULL then return x
10:     for c = 1 to T.R do
11:         if x.proximo[c] ≠ NULL then return x
12:     return NULL
```

O procedimento `DELETE` simplesmente atribui a `T.raiz` o valor retornado pelo procedimento recursivo `DELETE-RECURSIVE`. Este último recebe como argumentos a mesma *trie* T, um nó x, uma chave k e um inteiro não negativo d.

Se uma chave k não está associada a nenhum nó $x \in T$, então o procedimento não faz nada. Agora, se uma chave k está associada a um nó $x \in T$ que é interno, então simplesmente atribuímos a `x.val` o valor `NULL`. Agora, se x for um nó folha, então precisamos remover o nó x da *trie* T, bem como remover também todos os seus ancestrais que possuírem o campo `val` valendo `NULL`, até encontrarmos o primeiro ancestral com campo `val` diferente de `NULL`. Isto se faz necessário para manter a árvore otimizada, ou seja, com a quantidade de nós estritamente necessários. Caso não houvesse a remoção de nós, a árvore ficaria com nós "inúteis", o que compromete o uso eficiente de memória e processamento.

A cada chamada recursiva a `DELETE-RECURSIVE`, o valor de d, que inicialmente vale 1, é incrementado, até que o mesmo atinja o valor $|k| + 1$, ou até que x valha `NULL`. Caso $d = |k| + 1$ (linha 6), então atribuímos a `x.val` o valor `NULL`, e em seguida verificamos se existe alguma outra chave no sentido descendente (linhas 10-11). Caso exista pelo menos um ponteiro `x.proximo[`i`]` diferente de `NULL`, então isso significa que existem outras chaves que compartilham o mesmo caminho, então não removemos o nó x. Caso contrário, significa que não há mais nada "para baixo" na árvore, e neste caso é seguro remover o nó x. Por outro lado, se durante a recursão, atingirmos algum nó $x =$ `NULL`, então isso significa que a chave k não pode estar em T, pois nesse momento $d < |k| + 1$. Nesse caso, simplesmente retornamos `NULL`.

A cada passo da recursão, atualizamos o ponteiro `x.proximo[`c`]` (linha 8), onde $c = k[d]$ é o caractere associado ao nível da recursão atual.

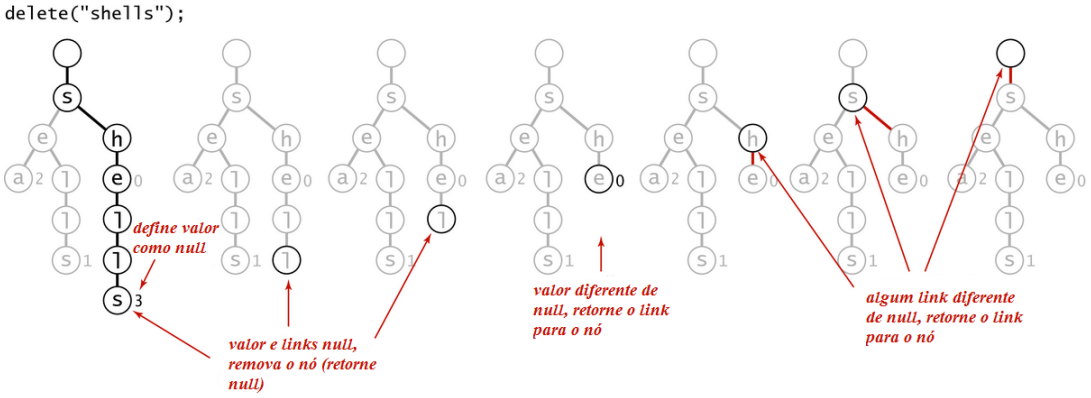

Figura 6.6: Funcionamento do algoritmo `DELETE` para a remoção da chave `shells` de uma *trie*.
Fonte: https://www.ime.usp.br/ pf/estruturas-de-dados/aulas/tries.html (adaptado)

6.4 Operações especiais

6.4.1 Obtendo todas as chaves com determinado prefixo

O procedimento `KEYS-WITH-PREFIX` recebe como argumentos uma *trie* T, um prefixo p e retorna uma fila q com todas as chaves presentes em T que contém o prefixo p. Este procedimento usa o procedimento recursivo auxiliar `COLLECT`, que será explicado em breve.

Algorithm 46

```
 1: procedure KEYS-WITH-PREFIX(T, p)
 2:     q ← ALLOCATE-MEMORY()
 3:     COLLECT(T, SEARCH-RECURSIVE(T.raiz, p, 1), p, q)
 4:     return q
 5:
 6: procedure COLLECT(T, x, s, q)
 7:     if x = NULL then return
 8:     if x.val ≠ NULL then q.enqueue(s)
 9:     for c = 1 to T.R do
10:         COLLECT(T, x.proximo[c], s+c, q)
```

A linha 2 do procedimento `KEYS-WITH-PREFIX` aloca espaço para uma nova fila q. Os detalhes operacionais foram abstraídos pelo procedimento `ALLOCATE-MEMORY`. Em seguida, a linha 3 invoca o procedimento recursivo `COLLECT`, que coletará as chaves que possuem o prefixo p, e finalmente a linha 4 retorna a fila q.

O procedimento `COLLECT` recebe como argumentos uma *trie* T, um nó x, uma string s e uma fila q. A string s está associada ao nó x, caso x não seja `NULL`. Em determinado ponto da recursão, se x for `NULL`, então o procedimento simplesmente retorna (linha 7); caso contrário, se `x.val` for diferente de `NULL`, então inserimos a string s, associada ao nó x, na fila (linha 8), e em seguida continuamos a recursão, buscando por filhos do nó x que sejam chaves de T (linhas 9-10). A string associada ao nó `x.proximo[c]` é $s + c$, onde s é a string associada ao nó x e $s + c$ é a concatenação da string s com o caractere c. Desse modo, a linha 10 continua a recursão no filho `x.proximo[c]` de x, tomando como argumentos a mesma *trie* T, o nó `x.proximo[c]`, a string $s + c$ associada ao nó `x.proximo[c]` e a mesma fila q.

Ao final de todas as chamadas recursivas, a fila q terá todas as chaves presentes em T que possuam p como prefixo.

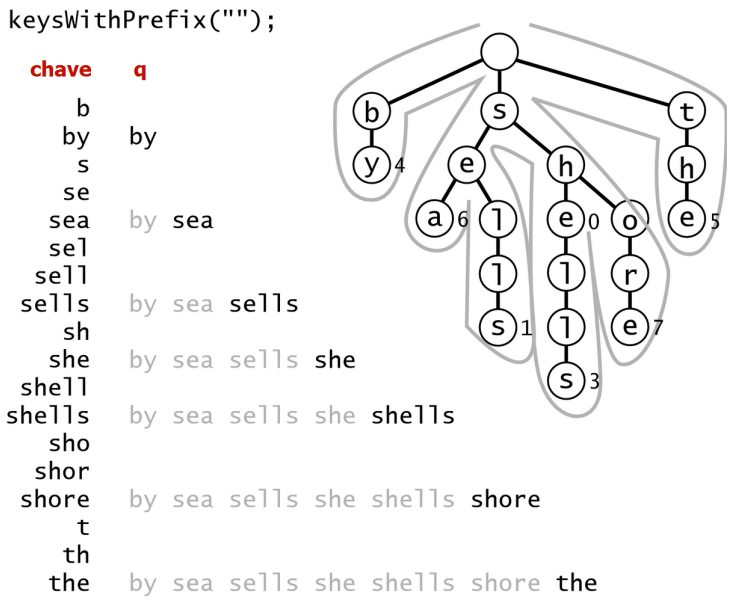

Figura 6.7: Uma chamada a `KEYS-WITH-PREFIX`, sobre uma *trie* T e tomando-se como argumento a string vazia, irá retornar todas as chaves presentes em T. A figura mostra como é realizada a busca em profundidade sobre T, e o estado da fila q em cada etapa.
Fonte: https://www.ime.usp.br/ pf/estruturas-de-dados/aulas/tries.html (adaptado)

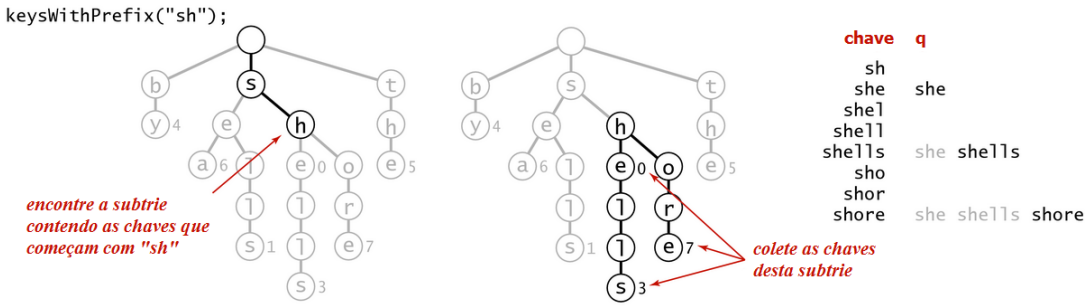

Figura 6.8: Uma chamada a `KEYS-WITH-PREFIX`, sobre uma *trie* T e tomando-se como argumento a string `sh`. Primeiro, localiza-se o nó x associado à string `sh`; em seguida, coleta-se todas as chaves presentes na subárvore com raiz em x. A figura mostra o estado da fila q em cada etapa.
Fonte: https://www.ime.usp.br/ pf/estruturas-de-dados/aulas/tries.html (adaptado)

6.4.2 Obtendo o maior prefixo presente numa *trie*, dada uma string

O procedimento `LONGEST-PREFIX-OF` recebe uma *trie* T, uma string s e retorna o maior prefixo de s que é chave de T.

Algorithm 47

```
1:  procedure LONGEST-PREFIX-OF(T, s)
2:      max ← -1
3:      x ← T.raiz
4:      d ← 0
5:      while x ≠ NULL do
6:          if x.val ≠ NULL then max ← d
7:          if d = |s| then break
8:          x ← x.proximo[s[d]]
9:          d ← d + 1
10:     if max = -1 then return NULL
11:     else return s.substring(0, max)
```

A linha 2 atribui a `max` o valor -1, provisoriamente. Esta variável irá guardar o maior índice tal que a substring $s[0, \texttt{max}]$ é chave de T. O loop *while* (linha 5) irá buscar o maior prefixo de s que é chave de T, enquanto x, que é o nó atualmente sendo examinado, for diferente de `NULL`, e enquanto $d < |s|$. Caso, ao final do loop *while*, este valor continue valendo -1, então simplesmente retornamos `NULL` (linha 10), já que não existe nenhum prefixo de s, nem mesmo a string vazia, que seja chave de T. Caso contrário, ao final do procedimento, é retornada a substring $s[0, \texttt{max}]$, podendo ser inclusive a string vazia, associada à raiz de T.

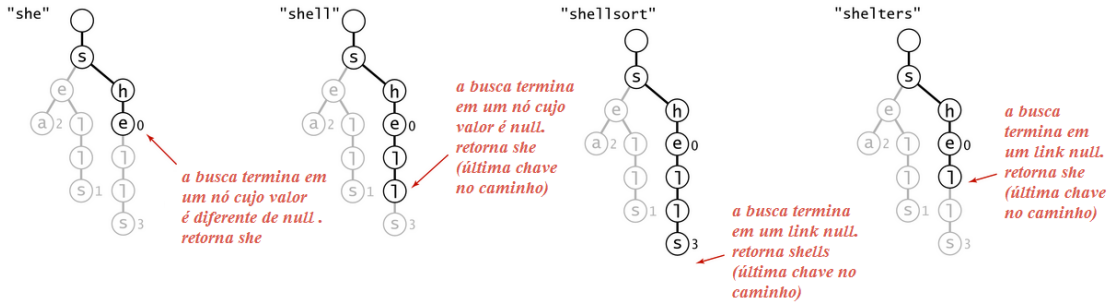

Figura 6.9: Diversos usos do procedimento `LONGEST-PREFIX-OF`, tomando-se diferentes strings como argumento. Fonte: https://www.ime.usp.br/ pf/estruturas-de-dados/aulas/tries.html (adaptado)

6.4.3 Obtendo todas as chaves que verificam determinado padrão

O procedimento `KEYS-THAT-MATCH` recebe como argumentos uma *trie* T, um padrão p e retorna todas as chaves de T que verificam o padrão p. Este padrão é composto por caracteres do alfabeto, juntamente com um caractere especial, chamado curinga. Toda vez que este caractere especial for encontrado em p, ele tentará "encaixar" qualquer outro caractere do alfabeto em seu lugar. Por exemplo, a string `abacate` verifica o padrão `a*a*a*e`, quando o caractere `*` é considerado como sendo o caractere curinga.

Algorithm 48

```
1:  procedure KEYS-THAT-MATCH(T, p)
2:      q ← ALLOCATE-MEMORY()
3:      KEYS-THAT-MATCH-RECURSIVE(T, T.raiz, p, q, 1, " ")
4:      return q
5:
6:  procedure KEYS-THAT-MATCH-RECURSIVE(T, x, p, q, d, s)
7:      if x = NULL then return
8:      if d = |p| + 1 then
9:          if x.val ≠ NULL then q.enqueue(s)
10:         return
11:     if p[d] = CURINGA then
12:         for c = 1 to T.R do
13:             KEYS-THAT-MATCH-RECURSIVE(T, x.proximo[c], p, q, d+1, s+c)
14:     else c ← p[d]
15:         KEYS-THAT-MATCH-RECURSIVE(T, x.proximo[c], p, q, d+1, s+c)
```

A linha 2 do procedimento `KEYS-THAT-MATCH` cria uma nova fila q, que conterá todas as chaves que verificam o padrão p. Em seguida, na linha 3, o procedimento recursivo `KEYS-THAT-MATCH-RECURSIVE` é invocado; é ele quem irá inserir as chaves que verificam o padrão p na fila q. Por fim, tal fila é retornada na linha 4.

O procedimento `KEYS-THAT-MATCH-RECURSIVE` recebe como argumentos uma *trie* T, um nó x, o padrão p, a fila q onde os resultados serão guardados, um contador d que controlará o nível máximo que a recursão irá atingir e uma string s, associada ao nó x (quando $x \neq \texttt{NULL}$). O procedimento, inicialmente, é invocado tomando-se como argumentos $x = \texttt{T.raiz}$, $d = 1$ e s a string vazia. Em determinada chamada recursiva, o procedimento verifica se $x = \texttt{NULL}$ (linha 7). Se for, então o procedimento simplesmente retorna. Caso contrário, o procedimento verifica se $d = |p| + 1$. Se isso

acontecer, e `x.val` $\neq$ `NULL`, então insere-se a string s, de tamanho $|p|$, na fila q. Por outro lado, se `x.val` $=$ `NULL`, então o procedimento retorna. Agora, se nenhum dos casos anteriores ocorreu, então a recursão deverá continuar "descendo" pela *trie*, e é verificado se o caractere $p[d]$ associado ao nível da recursão atual é o caractere curinga (linha 11). Se for, então a recursão continuará em cada um dos filhos do nó x (linhas 12-13). Se não for, então a recursão deve continuar apenas no filho adequado (linhas 14-15). Ao final de todas as chamadas recursivas, a fila q conterá todas as chaves que verificam o padrão p.

6.5 Usos práticos

As *tries* são utilizadas na prática nos seguintes contextos:

- Ordenação lexicográfica de um conjunto de strings.
- Implementação de uma árvore de sufixos, utilizada para indexar todos os sufixos de um texto.
- Utilização em motores de busca, no recurso autocompletar.
- Presente em mecanismos de roteamento para a internet, para armazenar prefixos de endereços IP.

CAPÍTULO 7

CONCLUSÃO

Nos capítulos anteriores, vimos algumas estruturas de dados avançadas e algumas de suas aplicações. Começamos estudando a estrutura de dados árvore B: um pouco de sua história, seus procedimentos elementares e finalizamos o capítulo entendendo que sua aplicação se dá em diversos ramos, desde implementação de sistemas gerenciadores de banco de dados a implementação de sistemas de arquivos eficientes para sistemas operacionais. Depois continuamos, estudando as árvores splay, e entendemos que essa estrutura é usada, entre outras coisas, para implementar *caches*. Em seguida, estudamos os heaps binomiais, e vimos que essa estrutura é importante, por exemplo, na implementação eficientemente do algoritmo *round-robin*, presente em vários sistemas operacionais, usado no controle eficiente de processos. Logo após, vimos os heaps de Fibonacci, de interesse majoritariamente teórico, cujo objetivo é melhorar o tempo de execução assintótico de vários algoritmos valiosos em ciência da computação. Por fim, vimos as *tries*, que são utilizadas para o processamento eficiente de strings.

Nossos estudos nos mostram que o uso de estruturas de dados avançadas é necessário à medida que os problemas se tornam mais específicos. Assim sendo, é importante que os programadores e cientistas da computação conheçam uma grande variedade dessas estruturas, não se limitando às estruturas ensinadas usualmente nos cursos de ciência da computação, para obter a maior performance possível em seus sistemas computacionais.

Algumas outras estruturas, que não foram abordadas neste trabalho, e que certamente merecem bastante atenção, são: *skip lists* (estrutura de dados probabilística; é considerada como sendo uma das mais importantes estruturas de dados em ciência da computação), árvores R (usadas para indexar informação multidimensional), e árvores AVL (objetivam minimizar o número de comparações no pior caso de uma busca cuja chave é equiprovável).

REFERÊNCIAS

[1] CORMEN, Thomas et al. **Introduction to Algorithms**, 2nd edition.
London: The MIT Press, 2001.

[2] **B-tree**. Wikipedia.
Disponível em: <https://en.wikipedia.org/wiki/B-tree>.
Acesso em 11 de fevereiro de 2022.

[3] **Splay tree**. Wikipedia.
Disponível em: <https://en.wikipedia.org/wiki/Splay_tree>.
Acesso em 11 de fevereiro de 2022.

[4] **Splay Tree**. Astik Anand.
Disponível em: <https://astikanand.github.io/techblogs/advanced-data-structures/splay-tree>.
Acesso em 11 de fevereiro de 2022.

[5] **Binomial heap**. Wikipedia.
Disponível em: <https://en.wikipedia.org/wiki/Binomial_heap>.
Acesso em 11 de fevereiro de 2022.

[6] **Fibonacci heap**. Wikipedia.
Disponível em: <https://en.wikipedia.org/wiki/Fibonacci_heap>.
Acesso em 11 de fevereiro de 2022.

[7] **Trie**. Wikipedia.
Disponível em: <https://en.wikipedia.org/wiki/Trie>.
Acesso em 11 de fevereiro de 2022.

[8] **Estruturas de dados: Tries**. Paulo Feofiloff.
Disponível em: < https://www.ime.usp.br/ pf/estruturas-de-dados/aulas/tries.html>.
Acesso em 11 de fevereiro de 2022.